TOWARDS PROVABLY CORRECT SERVICES:
AUTOMATED SERVICE COMPOSITION VIA SUPERVISORY
CONTROL SYNTHESIS

by

FRANCIS KWESI ATAMPORE

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
September 2017

Copyright © Francis Kwesi Atampore, 2017
Abstract

Service-oriented computing (SOC) is a distributed computing paradigm that is revolutionizing the development of software systems. Service-oriented architecture (SOA) provides a framework for realizing and implementing SOC. A Web service is a key concept for developing SOA applications that allows interoperability among distributed software applications deployed on different platforms and architectures, which is important for many electronic business applications. Web services enable organizations to carry out certain business activities automatically and in a distributed fashion.

However, in some circumstances, a single service is not able to perform a certain task and it becomes necessary to compose two or more services in order to complete it. Thus, a key research challenge in SOA is the problem of automated service composition. Several approaches exist that tackle the problem of automatic service composition; however, the task of generating provably correct Web service compositions still remains a challenging and complex task. The goal of this dissertation is to leverage the existing work on supervisory control to solve the problem of automated service composition with a focus on control and correctness.

Therefore, in this dissertation, we develop a novel formal framework for modeling Web
service compositions based on Supervisory Control Theory (SCT) of discrete-event systems. We model services that exchange messages and exhibit nondeterministic (runtime-dependent) behaviours based on runtime input. The objective is to synthesize a supervisor that interacts with a given set of Web services through messages to guarantee that a given specification is satisfied. The framework employs Labelled Transition Systems (LTSs) equipped with guards and data variables to model Web services and provides a technique to synthesize a controller. We model the interactions of services asynchronously and we use the guards and data variables to express certain preconditions which are then propagated from the system requirements through the overall composite service. The dissertation also provides a prototype implementation toolkit and an evaluation of the applicability of the approach using a number of case studies.

A key novelty of this work is the application of control theory to service-oriented computing and the incorporation of runtime input into the supervisor generation process.
Co-Authorship

All publication resulting from this thesis were co-authored with my supervisors Dr. Juergen Dingel and Dr. Karen Rudie. Portions of this work have been published and presented in the Proceedings of the 13th IEEE International Workshop on Discrete-Event Systems (WODES), Xi’an, China, 2016. In this publication, I am the primary author and conducted the research under the supervision of Dr. Juergen Dingel and Dr. Karen Rudie.
Acknowledgments

No one who achieves success does so without acknowledging the help of others.

The wise and confident acknowledge this help with gratitude

– Alfred North Whitehead

I am extremely fortunate and honoured to have Dr. Juergen Dingel and Dr. Karen Rudie as my PhD supervisors. I thank them for their time and support. In fact, I am indebted to them. They provided me with great direction by supervising, advising, and guiding me from the very early stages of this research. Dr. Juergen Dingel and Dr. Karen Rudie sacrificed their time to review countless numbers of drafts and attend research meetings; happy to answer any questions I put forth. I appreciate their detailed and valuable feedbacks. They provided me with financial support and were very sensitive to my cultural background. They ensured that I am content and comfortable throughout my PhD career made it an extraordinary experience. I especially enjoyed their ability to co-supervise me and provide me with their knowledge from the computing and engineering realm. They have demonstrated what it means to be effective supervisors and have affirmed reason why I am eager to join the PhD community.

I am also very grateful to my PhD defense examination committee: Dr. Stephane
Lafortune, Dr. Jenny Zou, Dr. Mohammad Zulkernine, and Dr. Robin Dawes, the chair (Dr. Chris Ferrall), for reviewing my PhD work and providing valuable feedback which help shaped the content of this thesis tremendously. With all my respect, gratitude and appreciation, I would like to deeply thank Dr. Dorothea for serving in my PhD supervisory committee and giving me the opportunity to give a quest lecture to her operating systems class for three consecutive years. Thanks to Dr. Pat Martin for his valuable comments and serving as the examiner for my PhD comprehensive exams. I cannot complete this paragraph without saying a big thank you to Dr. Khalid Elgazzar who devoted his valuable time and expertise to support me in the quest of finding a PhD research topic at the early stages of my PhD studies. I would like to acknowledge NSERC and Queen’s University for their funding support that made this research possible.

Many other people did not contribute directly to my research but helped remarkably along this very long journey. Thanks to my late father John Atampore who used the last of his resources to support my education. I would like to deeply thank my beloved mother, my sister and my four brothers for their endless love, support and encouragement. Thanks to Vanessa whom I met at Brock University while I was pursuing my Masters degree, I appreciate the countless amount of sacrifices and patience you have exercised throughout my academic career. I would like to express my appreciation to my colleagues from the DES Lab Behnam, Swapan, Tawhid, David and Usman for their comments and suggestions. Special thanks to MASE Lab members (Tuhin, Karolina, Amal, David, Gehan and Paul) for being amazing colleagues. I am thankful to the School of Computings administrative staff, especially Debby
Robertson for countless number of answers she provided to my questions relating to funding issues.
Statement of Originality

I hereby certify that the research presented in this dissertation is the original work of the author, conducted under the supervision of Dr. Juergen Dingel and Dr. Karen Rudie. Ideas and techniques published (or unpublished) that are not a product of my own work are cited in accordance with standard referencing practices, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.

Francis Atampore
September 28, 2017
Contents

Abstract i
Co-Authorship iii
Acknowledgments iv
Statement of Originality vii
Contents viii
List of Tables xi
List of Figures xii
List of Abbreviations xv

Chapter 1: Introduction 1
  1.1 Research Motivation ........................................ 2
  1.1.1 Web Services ................................................. 3
  1.1.2 Service Composition Problem .............................. 4
  1.1.3 Discrete-Event Systems Control Theory .................. 7
  1.2 Problem Statement ............................................ 8
  1.2.1 Challenges .................................................. 9
  1.3 Scope .......................................................... 13
  1.3.1 Proposed Approach ......................................... 16
  1.4 Contributions ................................................ 18
  1.5 Organization of Thesis ....................................... 21

Chapter 2: Background and the State-of-the-Art 24
  2.1 Service-Oriented Architecture ............................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1</td>
<td>Web Services as an Implementation for SOA</td>
<td>28</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Web Service Composition</td>
<td>31</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Service Composition: Industrial Standards and Related Technologies</td>
<td>34</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Syntactic-Web Service Description and Composition</td>
<td>35</td>
</tr>
<tr>
<td>2.1.5</td>
<td>Semantic Web Service Descriptions</td>
<td>37</td>
</tr>
<tr>
<td>2.2</td>
<td>State-of-the-Art: Automated Web Service Composition</td>
<td>40</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Formal Modeling Languages</td>
<td>40</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Algebraic and Logic Reasoning</td>
<td>47</td>
</tr>
<tr>
<td>2.2.3</td>
<td>AI Planning</td>
<td>51</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Workflows</td>
<td>59</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Graphs</td>
<td>61</td>
</tr>
<tr>
<td>2.3</td>
<td>Discrete-Event Systems</td>
<td>65</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Preliminaries</td>
<td>65</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Supervisory Control Theory of DES</td>
<td>66</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Application: Discrete-Event Systems and Software Modeling and Automation</td>
<td>69</td>
</tr>
<tr>
<td>2.4</td>
<td>Most Closely Related Work in DES</td>
<td>73</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Comparison with the State-of-the-Art</td>
<td>77</td>
</tr>
</tbody>
</table>

Chapter 3: **Supervisor Aware Service Composition Architecture (SASCA)**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Service Composition Problem</td>
<td>80</td>
</tr>
<tr>
<td>3.2</td>
<td>Service and Supervisory Control Theory Representation</td>
<td>83</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Specification of Services using the WS-BPEL and the WSDL Languages</td>
<td>84</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Formal Specification of Services: The Service Labelled Transition System (SLTS)</td>
<td>84</td>
</tr>
<tr>
<td>3.2.3</td>
<td>WS-BPEL as an SLTS</td>
<td>92</td>
</tr>
<tr>
<td>3.3</td>
<td>The SASCA Framework</td>
<td>93</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Controller Synthesis for Service Composition</td>
<td>93</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Asynchronous Communication and System to be Controlled</td>
<td>96</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Preprocessing Design Errors</td>
<td>98</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Composition Requirements</td>
<td>103</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Controller Synthesis</td>
<td>104</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Controllability</td>
<td>114</td>
</tr>
<tr>
<td>3.4</td>
<td>Composition Synthesis Algorithm</td>
<td>120</td>
</tr>
</tbody>
</table>
3.4.1 Discussion: Manual Steps and Consequence of Design Choices of Forcible Events .............................................. 142
3.5 Proof of Correctness and Completeness ........................................ 150
  3.5.1 Proof of Controller Existences ........................................ 150
  3.5.2 Minimally Restrictive Controller ....................................... 173
3.6 Analyzing the Computational Complexity ................................... 195

Chapter 4: SASCA Prototype Implementation .......................... 200
  4.1 SASCA Architecture ....................................................... 200
  4.2 Prototype Implementation ................................................ 203
    4.2.1 Representing Services in SASCA .................................. 204
    4.2.2 The SASCA Translator .............................................. 207
    4.2.3 Transformation Rules ............................................... 210
    4.2.4 Algorithm Implementations and Executing SASCA ............. 217

Chapter 5: Application and Evaluation ................................ 222
  5.1 Description of Experimental Setup ..................................... 223
  5.2 Case Study 1: Travel Reservation ...................................... 224
  5.3 Case Study 2: Virtual Heavy Duty Industrial Spare Parts Delivery Services ......................................................... 232
  5.4 Discussion ................................................................. 241

Chapter 6: Conclusions ...................................................... 243
  6.1 Summary ................................................................. 245
  6.2 Limitations and Future Work .......................................... 247

Bibliography .......................................................... 251

Appendix A: Algorithm 3.1 Java Source Codes for Chapter 4 282
Appendix B: Code Excerpts for Chapter 5 286
List of Tables

2.1 The Web Service Protocol Stack ........................................ 30

2.2 Application of Discrete-Event Systems to Software Engineering (the
direction of the arrow indicates the progressive use of DES to tackle
software engineering problems) ........................................... 74

3.1 WSDL Constructs [1,35] .............................................. 85

3.2 WS-BPEL Constructs [5,6] ........................................... 86

4.1 Logic operators used in guards ....................................... 207

5.1 Experiment Results .................................................. 230
List of Figures

1.1 ................................................................. 13

2.1 Where the topic of this dissertation lies with respect to state-of-the-art 25
2.2 Service-Oriented Architecture ................................. 28
2.3 Composition Models ........................................... 33

3.1 Available Component Services for Flight Reservation and Purchase System ........................................... 94
3.1 Available Component Services for Flight Reservation and Purchase System ........................................... 95
3.2 Asynchronous Parallel Composition ............................ 99
3.3 Unspecified-receptions and Non-executable interactions .......................................................... 102
3.4 Composition Requirements ...................................... 105
3.5 Plant $G_W$ .......................................................... 112
3.6 ................................................................. 117
3.7 Plant Service $G_W$ .................................................. 118
3.8 Target Service $T^W$ ............................................... 118
3.9 Flow-chart Describing the Interdependence of the Algorithms ...................................................... 121
3.10 Illustrative Example Using the Algorithm ....................... 144

xii
5.4 Portion of Controller for Travel Reservation . . . . . . . . . . . . . . . 229
5.5 Snapshot of Sequence of Interactions: Flow Trace Taken From the
Oracle Business Process Manager (Enterprise Manager) . . . . . . . . 233
5.6 Shipping Service . . . . . . . . . . . . . . . . . . . . . . . . . . . . 236
5.7 Snapshot of Sequence of Interactions . . . . . . . . . . . . . . . . . 240
5.8 Snapshot of Sequence of Interactions . . . . . . . . . . . . . . . . . 241

xiv
Chapter 1

Introduction

We are undergoing a major paradigm shift in software development as a result of the emergence of the World Wide Web. This development has brought about the Service-oriented computing (SOC) paradigm, which aims to encapsulate software components and expose them as services through network-accessible platforms, and language independent interfaces. According to Alonso and colleagues [4], service-oriented computing is one of the most promising state-of-the-art cross-disciplinary paradigm and software methodology. It is systematically revolutionizing the way software applications are designed, made, marketed and consumed. Service-oriented architecture (SOA) provides a framework for realizing and implementing SOC. Service-oriented architecture aims at developing simpler and low-cost distributed applications.

Services form the fundamental building block of SOC used to support the development of rapid, cost effective and easy interoperability of loosely coupled heterogeneous distributed applications [126]. Services are informally defined as autonomous,
self-describing and platform-agnostic computational entities that can perform various functions ranging from responding to simple requests to complex enterprise processes.

The vision underpinning SOC is very ambitious, and therefore also very attractive to any organization, particularly, the e-Business section (e.g., online service provision), the e-Government and the e-Science domains. Unfortunately, efforts to achieve the key principles of SOC have resulted in several challenging research issues, including service specification, service discovery, service composition, service execution, service monitoring and management.

The objective of this dissertation is to provide a supervisory control solution to the problem of automated service composition, namely, the problem of automatically combining a set of available services into a new one in order to satisfy a user request. More specifically, the overall goal is to demonstrate that supervisory control theory of discrete-event systems can be applied to simplify and automate the process of software development. We present an approach to the Web service composition problem with a particular focus on control and correctness.

1.1 Research Motivation

In this section, we present the motivation behind the problem that we investigate and study in this dissertation, namely, services and the problem of automated service composition.
1.1.1 Web Services

Web services provide a standard method of interoperability and communication between different software applications over a network running on heterogeneous platforms. A Web service can be seen as any piece of software that is available over the Internet and uses a standardized XML messaging system [25]. Web services allow applications deployed over the Internet to exchange and use information in a decoupled and reusable fashion.

In distributed systems, such as business, government and healthcare applications when the functional requirements increasingly become very complex and sophisticated, the task of finding a single Web service to satisfy a user’s request or specification becomes very difficult and in most cases impossible. Thus, one of the most important aspects of Web services is the ability to combine multiple services in a loosely coupled way in order to achieve more complex functionalities as “added-value services” [108]. Let us consider a classic motivational example as follows.

Example 1.1.1. Consider a group of families who want to go on their annual family fly and cruise vacation trip and want to make arrangements. The trip arrangements include the registration and the booking of a cruise package, a resort reservation, and a travel reservation to transport them to and from where the cruise will start. Suppose that there exist only a cruise package booking Web service, a vacation resort reservation Web service and a travel reservation Web service, which reside separately in an online repository that are accessible to the families. Now, it can be seen that none of these three Web services can satisfy all of the above client’s requests at once. Therefore, we want to combine the three Web services into a new
1.1. RESEARCH MOTIVATION

A composite service rather than implement a new one from scratch. In order to satisfy the client requests, one must ensure a correct orchestrating or coordinating of the activities among the three services. A correct execution of these three services will ensure that the families first book a cruise package by invoking the cruise package booking Web service, which is followed by an invocation of the vacation resort reservation Web service based on the outcome of the cruise package Web service. Once all the cruise packages and the resort reservations have been confirmed by the providers, next, the travel reservation service is invoked.

In the next section, we introduce the problem of automated service composition.

1.1.2 Service Composition Problem

In fact, the full potential of Web services can be fully achieved if we envision a collaboration between a community of numerous service providers and service consumers who interact in order to achieve certain business goals. Thus, services are combined into a business application (service composition) that specifies complex business logic in which execution of multiple services is coordinated. Service composition is a key driving force of SOA which seeks to provide the means to quickly create new services and applications (composite services) by “composing” the existing ones (atomic services).

More specifically, service composition comes into play when the current available services cannot satisfy a client’s request or provide the needed functionality; consequently, a composite service is used by combining parts of the available component services.
Service composition creates, selects and integrates preexisting services to develop value-added services and applications. Thus, it promotes rapid application development by reducing the cost and effort for the creation of new services. Furthermore, the output of a service composition (i.e., composite service) could serve as the input atomic service for further service compositions, promoting reusability. The benefits of Web services and their compositions are not only appealing to the SOC paradigm, but are very attractive and promising in the area of Cloud Computing, which is one of the newest paradigms gaining a considerable amount of momentum over the past few years for the delivery of sophisticated on-demand computing infrastructures, and services over the Internet [32]. It is no surprise that a colossal amount of development in terms of technologies and standards in SOA seeks to address the problem of service composition, and it is one of the most actively researched areas in SOA and Cloud Computing. In addition, service composition is of considerable interest to both industry and academic research. More interestingly, just this year, the article (a research manifesto) by Bouguettaya et al. [26] published in Communications of the ACM Journal highlights service composition as one of the research challenges that needs to be tackled in the next ten years.

In order to achieve the main objective of Web services and their compositions, one of the cornerstones and the most important objective of this paradigm is to ensure a correct and efficient composition of Web services which guarantees that a given specification of a given task is met. Thus, when services are combined a key challenge is to guarantee the correctness and robustness of the composition. For example, in Example 1.1.1, in order to achieve a meaningful and effective composition, it is
imperative to ensure the proper and the correct coordination of the Web services involved.

Several approaches have been proposed in an attempt to address the problem of automatic service composition. Most of these techniques are motivated by the research in AI (Artificial Intelligence) planning and cross-enterprise workflow techniques [24, 74, 88, 92, 134, 142, 183]. However, the task of generating provably correct Web service compositions still remains a challenging and complex one. The challenge to ensure the correctness of an automatic service composition is highlighted in a survey of Web service composition approaches presented by Ter Beek et al. [15]. The survey indicates that “the main problems with most practical approaches to Web service composition are the verification of (behavioural) correctness of Web service compositions” [15]. In other words, the way the problem of service composition is addressed has been a mostly ad hoc, error-prone and manual process. In order to synthesize a correct composition, some existing techniques decompose the problem into two phases. The first phase computes a composition and then the second phase performs a verification on the generated composition by resorting to techniques such as model checking or replanning to reconfigure the system when a failure has already occurred. In this era, where multi-million dollar business transactions are carried out over the Internet each day, an error in a system design could potentially cause drastic losses and ad hoc repairs after failures are not acceptable. The goal of this dissertation is to leverage the existing work on the supervisory control problem to propose an approach to automatic service composition that guarantees the correctness of the resulting composite service.
1.1.3 Discrete-Event Systems Control Theory

In this dissertation, we propose a correct-by-construction approach to address the problem of automatic service composition by applying the supervisory control theory of Discrete-Event Systems (DES). Discrete-Event Systems control theory is a branch of control theory that models behaviours as sequences of discrete events instead of as continuous functions of time. The classical Ramadge-Wonham approach to the supervisory control problem (SCP) [33, 172] is defined as follows: given a plant $G$ modeled in the form of a state transition system which captures the behaviours of the process to be controlled according to some possible events, given a set of specifications $L$ which describes the legal sequences of events of the plant, synthesize a supervisor $S$ so that $S$ restricts $G$ in such a way that all its executions satisfy $L$ and such that $S$ is minimally restrictive.

The DES control methods fit well into the problem of automatic service composition if the problem is reduced to observing events of the system and restricting its behaviour to specific sequences. Hence, we can apply existing DES techniques and algorithms to address the problem of automatic service composition.

Compared to other service composition approaches, Supervisory Control Synthesis has several benefits: It results in a correct-by-construction control synthesis, and the generated controller is minimally restrictive by preventing a system’s behaviour only if it violates the system requirements. It also relies on automata theory to provide a well-defined syntax and semantics for modeling systems which can be very useful for specifying services. In addition, DES provides a standard way that can be used to model various business logics and requirements in a dynamic environment.
Supervisory control theory has been applied to generate concurrency control code in multithreaded programs. Notable results can be found in the work by Dragert et al. [7, 53] and the in work by Wang et al. [166, 168]. It is based on this background that we propose to apply SCT to service-oriented computing. To the best of our knowledge, our work is the first of its kind to apply SCT to deal with the problem of automated Web service composition.

1.2 Problem Statement

The problem we address in this dissertation is to develop a correct-by-construction formal framework which we call the Supervisor Aware Service Composition Architecture (SASCA), for modeling Web service composition based on Supervisory Control Theory (SCT) of Discrete-Event Systems (DES). That is, we hypothesize that supervisory control theory of discrete–event systems can be leveraged to better address the problem of automatic service composition with respect to correctness and control. In this thesis, we develop a formal framework by building on supervisory control theory to solve the problem of automatic service composition. In the framework, the problem of automatic composition synthesis is seen as a problem of supervisor synthesis in DES. As part of our work, we determine whether DES is beneficial to the SOA community or not. Informally, the composition problem that we solve in this thesis is as follows:

Given a set of available services $G_W^1, G_W^2, \ldots, G_W^n$ and a set of specifications $T_W$ representing the goal (or desired) service over the same environment (same set of atomic actions), we would like to construct a controller $C$ that is nonblocking
which interacts with the available services to satisfy the specification $T^W$. Thus, $C$ serves as a controller that restricts the system in such a way that all its executions satisfy $T^W$ and so that $C$ is minimally restrictive. In addition to requiring that the generated controller satisfies the nonblocking criterion, the controlled system is also free of errors that may result from communication among component services.

1.2.1 Challenges

- Automated service composition is a problem in software engineering, whereas supervisory control theory is rooted in electrical engineering. Thus, we have to find out how we can best adapt SCT to this new setting. For example, in the software engineering domain what we call actions are known as events in supervisory control theory.

- Standard supervisory control theory has a well-established body of knowledge, however, we cannot use only standard supervisory control theory since Web services and the problem of automated service composition have additional constraints that need to be taken care of. The use of finite state automata for modeling in supervisory control theory abstracts away a lot of very important details needed to model services and their compositions. Therefore, in order to model services we need to provide support for the following:

  - Message exchanges, data, variables, and parameters: The main characteristic of Web services is their ability to explicitly exchange messages, and to represent and manipulate data variables and parameters. Standard supervisory control theory does not provide direct support for explicit message
exchange involving data. Hence, to make SCT beneficial to the service computing domain, we need to represent and incorporate messages, data and variables or data parameters during the modeling process.

- Conditions (guards on transitions): Another equally important challenge is to deal with preconditions and postconditions in specifying services and their compositions. Since standard SCT uses finite state automata for modeling, it does not make explicit use of conditions on data variables which are constituents of services. In general, a precondition expresses a property that needs to be satisfied before some operation can be executed. For example, in numeral terms, a precondition $\text{balance} \geq 1000$ may impose a restriction on the amount of cash in a given bank account, whereas a postcondition $\text{balance} = \text{balance} - 400$ may allow a withdrawal of money from the account. The challenge here would then be how to place a restriction on the kind of data (such as the set of values a variable takes or what data parameters can be exchanged among services) that can be sent and received. Having said this, there exist supervisory control theory approaches based on extended finite state machines [113,123,157,176] which capture the notion of guards, but these techniques still fall short in the model we consider. We provide more details on this issue in Section 3.2.

- Information or data that may not be known until runtime (runtime-dependent):

The challenge is how to deal with runtime information. The use of runtime information in our model is inspired by the fact that the services we model
1.2. PROBLEM STATEMENT

are nondeterministic (runtime-dependent) and only partially controllable. That is, the outputs of a service cannot be predicted a priori and some of the internal computations of a service are hidden from other external services. For example, the information about whether there are still seats available on a flight cannot be known until runtime or the data on whether there is enough cash in a customer’s bank account will only be available at runtime. Due to nondeterminism and the black-box nature of some of the events in our model, the use of classic supervisory control theory in many cases will result in an empty controller or an overly restrictive one.

Let us illustrate the above discussion with the following example. Consider Figure 1.1, which models an airline reservation system and its specification. For now, we are not interested in the formal details of this example; we deal with these technicalities in a subsequent section (Section 3.2.2). Figure 1.1(a) represents an airline system (plant) which upon accepting a request, checks the availability of a flight and returns an offer if it is available. Assume that Figure 1.1(b) is the system requirement to be met. The specification (Figure 1.1(b)) models the same behaviour as the plant except for the transition from state $t_4$ to $t_5$ where there is a restriction on what branch to take based on the value of the variable. More specifically, the specification restricts the values of $av$ to KLM or Delta. That is, at state $s_4$ the specification allows the plant to transition to state $s_5$ only if the value of the variable $av$ is either KLM or Delta; anything else is not allowed. Assume the transitions labeled
checkAirlinesAvail(date, loc :: av) and processBooking() are not controllable\(^1\). The use of the classic DES will never allow the plant to reach state \(s_4\) since checkAirlinesAvail(date, loc :: av) and processBooking() are uncontrollable. This implies that the system will receive a request but will never return a response, which does not make much sense in the service domain. In other words, it will be overly restrictive if the classic DES is applied directly. The use of supervisory control based on extended finite state machines would also not work, since the value of the variable \(av\) is unknown and cannot be determined until state \(s_4\) has become active. The event checkAirlinesAvail(date, loc :: av) is assumed to be black-box\(^2\), i.e., the effect that it has on variables is not known.

Intuitively, in order to model services and their compositions using supervisory control theory, we need to provide support for (i) message exchanges, (ii) data and variables, (iii) guards, and (iv) information or data that may not be known until runtime. Standard supervisory control theory may not be sufficient to model services or may lead to an overly restrictive controller.

- Also, since we will be using transition systems for modeling, SCT can lead to a space explosion problem, which we have to take into account to some extent. Even though the objective of this thesis is not to address the space explosion problem, we still have to keep it in mind when deciding on what techniques to choose during the development of a solution to the service composition problem.

---
\(^1\)In standard SCT, an event that is not “controllable” is one that cannot be prevented from occurring.
\(^2\)See Section 3.2.2 for further details.
1.3 Scope

The methodology and contributions presented in Chapter 3, Chapter 4 and Chapter 5 of this document provide a comprehensive discussion on the scope of work. In summary, the dissertation examines the use of supervisory control theory to solve the problem of automated service composition within the following scope:

- **Formal Service Composition Framework**: This thesis investigates and develops a compressive formal framework for the problem of automated service composition by tackling the following aspects.
  - Services description and representation: here we look at how Web services can be represented using a formal language that clearly and fully specifies

![Diagram](a) $S_1$

![Diagram](b) $S_2$
services including how this formal language can be derived from industrial languages such as Web Services Business Process Execution Language (WS-BPEL) [5,6] and also translated back. That is, how to formally specify Web services so that they can be modeled as a discrete event system, and all the relevant aspects of Web services are supported. In doing so, the model takes into account the properties of Web services such as message exchanges, data variables and parameters, and conditions. That is, we answer the question of how Web services can be specified so that supervisory control synthesis can be applied.

- The representation of functional requirements is one of the most crucial concepts when developing a complete and sound solution to service composition. Most existing approaches express functional requirements in terms of only input and output parameters without specifying behavioural constraints, which is insufficient. Our objective is to clearly specify functional requirements taking into consideration the behaviour of services.

- Solution to the problem of automated Web service composition: In this dissertation, we formalize the problem of automated Web service composition using supervisory control synthesis constructs, and then we formulate the composition problem as a control problem and finally provide a formal solution to this problem. In addition, we examine how to orchestrate data and control flow requirements after the generation of a composition.
1.3. SCOPE

- Minimally Restrictive: the solution to the problem of automated Web service composition that we seek to provide also takes into account the minimally restrictive property of the generated controller.

- Refinement of Web Service Composition Specification: Using our approach, it is possible that no composition exists to satisfy a given specification due to incomplete specification or the specification provided by the composition designer results in a failure of a composition. It may be of interest to find an approximate “solution”. More precisely, in the case where a composition does not exist, one may be interested in understanding which parts of the specification cannot be realized and which can. Realistically, the composition designer would have to reconstruct the specification and find another specification to do the work. As part of the composition synthesis algorithms that we provide, we seek to aid the reconstruction of a given specification when a composition cannot be met due to a failure caused by the specification using an iterative refinement technique that refines the specification until a composition can be generated. In the worst case, the refinement will result in an empty specification which will lead to an empty composition.

- Web service composition algorithms: The thesis develops a set of formal algorithms for the proposed controller synthesis framework to solve the problem of automated Web service composition.

\[3\text{In SCT a controller is minimally restrictive in the sense that it only disallows transitions that must be disallowed.}\]
• Correctness and completeness of the approach: In order to establish the theoretical correctness of our approach, we formalize the problem in a number of theorems and prove them.

• Prototype implementation: The objective of this work is not only to develop a formal technique, but to also provide a practical implementation (tool) realizing it.

• Evaluation of the proposed approach: The dissertation seeks to provide an experimental validation of the proposed approach based on its applicability and certain standard performance criteria (i.e., computation efficiency and effectiveness). This evaluation is performed on well-known examples. We determine the strengths and the weaknesses of our approach.

1.3.1 Proposed Approach

The approach we propose employs Labelled Transition Systems (LTSs) equipped with guards, and data variables to model a given set of Web services specified as WS-BPEL [6]. To this end, we provide an SCT modeling formalism based on the variant LTSs, and then we describe a novel technique to synthesize a composition satisfying a given functional requirement (data and control flow) also specified as LTS extended with guards and data variables. That is, the problem of synthesizing a composition can be reduced to the problem of supervisor synthesis when the available services and a goal specification represent the plant and the legal language (desired behaviour), respectively, in SCT. In this way, the problem of orchestrating data and control flow requirements can be achieved using the notion of controllability in SCT where the
supervisor enacts control by disabling and enabling certain actions in order to enforce the given goal. The inputs to the system are the set of Web services specified as a WS-BPEL and the requirements specified as a variant LTSs. Internally, we represent these Web services using a variant of LTS. Next, the proposed supervisory control framework based on LTSs is applied to synthesize a controller that ensures that the given composition requirements are satisfied. The generated controller’s transition system is translated back to WS-BPEL.

Our approach lies between the event-based supervisory control [172] and the state-avoidance control problem [68]. However, our approach extends the basic control capabilities in these approaches as follows. Apart from the supervisor being able to prevent certain events from occurring by properly disabling and enabling controllable events, the supervisor is also able to prevent the system from reaching certain sets of states designated as forbidden states by using runtime information of variables in the system. This is due to the fact that some of the events in our model are black-box (atomic actions) in nature and may exhibit nondeterministic properties. We deal with this nondeterminism through model refinement and the adaptation of event enforcement supervisory control theory [50]. Therefore, the generated supervisor not only restricts the behaviour of the plant, but also has the ability to actively enforce certain events. In addition, the generated supervisor is able to restrict the system by assigning stronger guards to data variables. This allows us to control the data a service can send or receive.
The research in this dissertation advances the current state-of-the-art in automated Web service composition by developing a comprehensive formal control synthesis framework based on supervisory control theory with a focus on control and correctness. In general, this research seeks to apply supervisory control theory to the problem of automated Web service composition. In addition, in the course of applying supervisory control theory to deal with the problem of automated Web service composition, it has also resulted in some contributions in the DES domain in which the standard supervisory control theory has been extended in order to model services and their compositions. More specifically, the contributions of this dissertation are detailed as follows:

1. **Providing a Novel Formal Supervisory Control Theory Framework for the Problem of Automated Service Composition**: We provide a novel supervisory control framework for automated composition of services, which uses Labelled Transition Systems (LTSs) with guards and data variables to model a given set of Web service specifications in industrial standard languages such as WS-BPEL. The framework proposed in this thesis has the following features:

- Support for behavioural specification of services: We provide insight into how to express and define functional specification in terms of data and
control flow requirements for a composition problem. Most existing approaches express functional specifications in terms of only input and output parameters without specifying behavioural constraints, which is insufficient. Our approach clearly specifies functional requirements taking into consideration the behaviour of services.

- Guaranteed Correctness: We provide a solution to the automated service composition problem by first formalizing the problem as a supervisory control theory problem and we provide a formal solution to this problem, in which the generated supervisor can be shown to be correct, i.e., enforce the specification. This technique also involves augmenting the generated controller, such that it is capable of enacting control based on information that is only available at runtime. We demonstrate the correctness of our approach by formulating the composition problem into theorems and providing proof of these theorems. First, we prove the existence of a controller using our approach, by showing that given a set of available services $G_W$ and a goal service $T_W$, there exist a controller $C$ such that when the plant is coupled with this controller it satisfies the specification. Second, we show that the controller generated using our approach is minimally restrictive, and finally, we provide various analyses on the computational complexities of the algorithms.

- Guaranteed Minimal Restrictiveness: The formal approach we present to generate a controller ensures that the controller is minimally restrictive. That is, given the set of available services $G_W$ and a specification $T_W$, the
controller generated for these inputs is minimally restrictive.

- Provides a Technique for Reconstructing of Specification to Allow for Composition: We provide a technique that enables iterative refinement of a specification to generate an alternative specification when a failure in a composition is due to incomplete specification or when it is not possible to compose the given services such that the specification is met.

2. **Presenting an Extension of Standard Supervisory Control Theory:** One of the major contributions of this thesis is the extension of the standard supervisory control theory in two major ways. Firstly, we extend SCT to be able to handle data and messages, variables and conditions. Secondly, we extend the standard SCT with runtime capabilities. That is, we integrate runtime input into the supervisor synthesis process such that the generated controller is able to enforce control based on the runtime information. Third, we develop a set of algorithms based on the formalism provided to generate a controller satisfying a given functional requirement also specified as LTSs. Beyond the standard disabling and enabling of events, the generated controller in our framework has the ability to enforce certain events based on runtime information to drive the system towards its goal. In addition, the controller is able to impose restrictions on the kind of data or variables that can be sent or received by the services. This construction includes the automatic generation of stronger guards or conditions which impose restrictions on which path to take during execution. These algorithms can be implemented in any programming language.
3. *Presenting a Prototype Implementation of our Automated Web Service Composition Framework into a Tool Support:* One of the objectives of this dissertation is to provide a tool to support the use of our technique, thus, we provide an implementation of the proposed techniques in the SASCA framework. The implementation involves the following: (i) the translation of industrial Web service standard such as WS-BPEL in a formal language suitable for control synthesis, (ii) the implementation of all the proposed control synthesis algorithms, (iii) the incorporation of runtime information into the supervisory control synthesis, and (iv) the translation of the formal language used in the synthesis back to the its WS-BPEL equivalent.

4. *Presenting an Evaluation of the Applicability and an Empirical Study of our Service Composition Technique:* We provide an experimental study of our implementation in terms of applicability of our approach and standard performance criteria including computational efficiency and effectiveness. We use well-known examples and case studies to demonstrate the applicability of the tool as well as its performance. We deploy and execute these case studies on the Oracle Business Process Manager Engine [41] and observe how the services interact with the generated controller. Based on the results of evaluation, we highlight the benefits of applying supervisory control theory to deal with the problem of automated Web service composition.

1.5 **Organization of Thesis**

The structure of the remainder of the dissertation is the following:
• Chapter 2 consists of two main sections, in the first section, we present briefly some of the basic concepts, and industrial standards and technologies for Web services and service compositions. We present a short analysis of the main architectures for Web service descriptions through related work. The second part of this chapter presents the literature review. This part provides an extensive overview of the state-of-the-art in automated Web service composition. It is followed by a literature review of relevant related work in DES and finally, we end the chapter by introducing supervisory control and the problem of automated service composition. Moreover, we provide a comparison between these approaches and point out some of the limitations of these approaches with respect to the approach presented in this dissertation.

• Chapter 3 presents the heart of this dissertation. It starts by presenting the formal language and various definitions which serve as the basis for our formalism. It then introduces the new service composition framework using the formalism provided earlier. To this end, we provide new definitions of controllability and the notion of an optimal controller, and then we formally introduce the composition problem and its solution. Next, we give the new set of composition synthesis algorithms and examples. The chapter also presents the main theorems of the thesis and their proofs. Additionally, this chapter provides a brief analysis of the computational complexities of the algorithms.

• Chapter 4 presents an implementation of the proposed technique. The SASCA system for automated Web service composition is described. The chapter describes the architecture of our system and its implementation details.
implementation involves the following main stages: (i) a transformation of WS-
BPEL and the Web Services Description Language (WSDL) [35] to a service
labeled transition system (ii) implementation of the controller synthesis algo-
rithm, and (iii) transformation of a service labeled transition system back to
WS-BPEL and its associated WDSL files. In addition, it discusses the compiler
and the transformation rules in detail.

- Chapter 5 provides an experimental evaluation of our approach. A brief dis-
cussion of Oracle SOA suite and its WS-BPEL engine is presented. The chap-
ter uses well-known case studies in the area of service composition to demon-
strate the applicability and computational complexities of our approach. It
also presents the results from experimenting with our approach in terms of
qualitative and quantitative analyses.

- Chapter 6 provides some concluding remarks of the dissertation and an out-
line of the major contributions of this thesis. This chapter ends with some
recommendations for future work.
Chapter 2

Background and the State-of-the-Art

The problem we address in this dissertation is deeply rooted in two major areas: automatic service composition and DES. Therefore, in order to facilitate the understanding of this work, we first present briefly some core concepts, industrial standards and technologies for Web services and service composition. Next, we present a survey on the state-of-the-art in automated Web service composition. This survey is followed by an overview of the relevant related research that applies DES to solve software engineering problems. Finally, we introduce our work and present the research most closely related to our work, that is, the combined work on both service composition and DES. Additionally, we give a comparison between our approach and the existing techniques. The Venn diagram in Figure 2.1 depicts where the research presented in this thesis lies with respect to the existing literature.
2.1 Service-Oriented Architecture

Service-oriented architecture forms the core implementation of SOC [57]. Service-oriented architecture presents an approach for reorganizing and developing distributed software systems by providing services to end-user applications, which provides interface descriptions that can be published and discovered. Service-oriented architecture consists of a collection of applications that expose and consume functionality as a service using contracts and messages [40].

According to the World Wide Web Consortium (W3C) Working Group, SOA is defined as follows.

Definition 2.1.1. Service-oriented architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations [25].
A modern definition of SOA given by Erl is as follows:

**Definition 2.1.2.** *Service-oriented architecture is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web services technology platform, SOA establishes the potential to support and promote these throughout the business process and automation domains of an enterprise [57, 58].*

A typical example of SOA implementation using Web services are Web applications which invoke another Web services to achieve a business goal. According to the text in the reference [55,57], the key principles of SOA are as follows:

- **Reusability:** a service logic can be divided into different granularities, which can be used by multiple processes, therefore promoting reuse.

- **Autonomous:** the business logic provided by each service is independent of other services. That is, a service has control over the logic it encapsulates and provides a business function that is independent of other services.

- **Contract-based:** services adhere to a set of communication agreements. Interface specifications are used to precisely define interfaces and policies.

- **Coupling:** the level of service dependencies is strictly limited, this dependency reduces the need for a service to modify its contracts if there is a change in the implementation.

- **Platform-independence:** both the consumer and SOA service systems can be on any platform that supports the service transport and interface requirements.
- Discoverability: services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

- Statelessness: services minimize retaining information specific to an activity.

- Composability: collections of services can be coordinated and assembled to form a composite service.

- Abstraction: the main logic of a service is hidden from any external agent and only reveals what is specified in the service contract.

The basic SOA framework consists of three major roles. These are (i) the service requester (the service client), an entity that locates and invokes the services in order to fulfill some desired goals; (ii) the service provider which creates and provides a service as well as publishes the service interface and its access information to the service registry (e.g., an organization) and lastly, (iii) the service registry which is a subject that acts as a repository of services; where services are published and discovered by service providers and service clients, respectively. When a provider wants to make a particular service available, it publishes the service interface description and its access information in the registry by specifying how to invoke it (e.g., URL, names, protocols). This specification includes information about the service itself. A client who wants to use a service to satisfy a particular goal locates a suitable service in the repository using a find operation and then binds (interacts) to the service provider in order to invoke the service. The basic model of SOA is illustrated in Figure 2.2. Since the implementation of SOA is not limited to a specific technology, several technologies have been proposed to model SOA [57]. These include Web
2.1. SERVICE-ORIENTED ARCHITECTURE

![Service-Oriented Architecture Diagram](image)

Figure 2.2: Service-Oriented Architecture

services [25], REST [11], DCOM [164], CORBA [120] and RPC [38]. The research in this dissertation is not tied to any specific SOA implementation, however, for the sake of simplicity, we stick to Web services and their related terminology, standards and specifications.

2.1.1 Web Services as an Implementation for SOA

As mentioned above, SOA is widely implemented using Web services [126]. The W3C [25] defines a Web service as follows:

**Definition 2.1.3.** A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using Simple Object Access
Protocol (SOAP) messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services only define the way components interact and do not impose any restrictions on the underlying implementation technologies and platforms. From Definition 2.1.3, it can be deduced that a typical Web service is a service that is (1) self-describing using XML format, (2) easily found using a simple find operation, (3) available over the Internet, (4) capable of exchanging messages using an XML messaging system, and (5) is not restricted to a specific platform. A standard Web service uses XML to represent data, SOAP [107] to transfer a message over the Internet using protocols (such as HTTP). It uses a WSDL specification to describe the availability of a service. The Web service protocol stack [171] is a developing set of protocols, technologies and standards which is used to specify, discover and implement Web services. Table 2.1 depicts the major class of protocols and standards that make up the stack. We provide a short review of the relevant protocols and standards in Section 2.1.2.

The major benefits of using Web services for SOA development include the following [55,57]:

- Web services promote loose coupling and better application integration: service-oriented architecture applications built using Web services can restrict the degree of dependency between each individual component. In this way, each unit operates independently from other units, thereby enhancing the adaptability and interoperability of the SOA applications far beyond that of traditional applications.
Table 2.1: The Web Service Protocol Stack

- Web services allow for application reusability: Web services enable service providers to easily build more sophisticated applications by reusing the logic of existing SOA applications. This reusability will consequently reduce the time between software development and implementation phases.

- Web services are designed to improve scalability and testability, thus the overall return on investment is high.

- Web services encourage encapsulation of business applications: Web services give service providers the ability to hide the implementation logic of their applications from service consumers or other services that invoke them. If a service consumer or an external service wants to execute a service, they are
only presented with the service’s interface and capabilities. This guarantees that they are not aware of how the service performs its internal computations.

2.1.2 Web Service Composition

As mentioned in Chapter 1, Web service composition is the process of synthesizing a more complex (composite) service by integrating existing Web services to satisfy a given request that cannot be fulfilled by a single pre-existing Web service. The key activities of a Web service composition are as follows: (1) in the beginning Service Discovery is used to find a set of Web services from a given repository (e.g., UDDI) that provides a given user functionality; at this stage, one must also deal with how to locate and select the discovered services; and (2) once the services have been identified and made available, next service composition combines these services to satisfy a given request. In this dissertation, our main focus is service composition. We assume that the services to be used for composition have already been discovered and made available, hence the problem of service discovery is not dealt with in this dissertation.

Orchestration and Choreography Service Composition Models: In practice, we can differentiate between two main concepts of modeling Web service composition [10, 20, 84, 90, 126, 130]. These are orchestration and choreography. Orchestration describes how multiple services involved in a composition can interact at the message level (exchanging messages) including the business logic and execution order of their interactions from the perspective of a single endpoint (i.e., the orchestrator). In terms of orchestration, one of the business participants involved
in the process always acts as a controller, which controls the interactions of the business-processes [126, 127, 130, 170]. It refers to an executable business process that can interact both internally and externally with Web services. The commonly used industry standard for modeling orchestration is WS-BPEL [5, 6].

A number of papers [18, 79, 94] distinguish between composition synthesis and orchestration. The authors of these papers propose that composition synthesis is concerned with synthesizing a new composite service from a given set of available services and a client request. However, the problem of orchestration deals with coordinating the various component services, and monitoring control and data flow requirements among the component services; in order to guarantee the correct execution of the composite service. This coordinating is done using the specification of the composite service synthesized in the composition synthesis phase.

On the other hand, choreography provides a global view of message exchanges and interactions that occur between multiple process endpoints, rather than a single process that is executed by one party. Thus, choreography is more collaborative (i.e., peer-to-peer architecture) and provides a way in which each participant can clearly define its part of interactions and rules [49, 126, 127, 130]. Web Services Choreography Description Language (WS-CDL) [87] is the language that is widely used in specifying choreography. Figure 2.3 gives a pictorial view of the differences between the two models.

Even though several works differentiate and implement orchestration and choreography separately, it is widely accepted that both orchestration and choreography
can co-exist within a unified framework, which indeed provides a complete representation.

**Static and a Dynamic Service Composition:** Another major classification of Web service composition methods is a static and a dynamic service composition. In a static Web service composition, the services, activities, and workflows to be composed are determined at design time. The composition is performed manually in such a way that the execution of Web services are executed one by one in order to achieve the desired goal. This construction implies that the generated composition is fixed based on the given requirements and cannot be changed during the execution of the composition, even when there are errors. However, in a dynamic Web service composition, the services to be composed are decided at runtime. That is, service composition is performed autonomously when a user queries for a Web service at runtime. In this case, the user is required to specify several constraints in order to invoke a service [73,97].
**Manual and Automatic Service Composition:** One should also note that, the service composition process can be performed either manually or automatically (or semi-automatically in some cases). In manual service composition, the process of specifying an abstract high-level composition task, the designing of the workflow and the searching and binding of Web services are all done manually by the developer. Manual service composition is difficult, time-consuming and error prone. Farooq’s Masters thesis [62] describes an approach for achieving a manual service composition in cloud based systems. In contrast, automated composition attempts to free the developer from the burdens of manual service composition with the help of automated tools. It involves minimal human intervention. Automatic service composition is the main focus of this work.

### 2.1.3 Service Composition: Industrial Standards and Related Technologies

Several industrial research efforts have made a lot of progress in shaping the development of the description of services and the problem of automatic service composition. Thus, our survey would be incomplete without mentioning industrial contributions. Therefore, in this section, we present the relevant state-of-the-art in industrial standards, and technologies geared towards the service descriptions and the composition of services.

Very big corporations and consortia, such as Microsoft [39], IBM [80], Oracle Corporation [42] and W3C [162], just to mention a few, are working on developing tools and providing platforms to automate the description of services and service
composition. Although industrial standards offer limited support for connectivity and QoS composition, they do not provide any direct means of dealing with the formal correctness of a service composition and hence, pave the way for resorting to formal methods approaches [15]. Standard languages for modeling services and service composition can broadly be classified into two main groups, namely, syntactic (XML-based) and semantic (ontology-based) languages. We only discuss a few of them here. However, we provide more details of WSDL and WS-BPEL in Chapter 4 as these languages are used in our framework.

2.1.4 Syntactic-Web Service Description and Composition

Basically, a Web service [117, 181] is a collection of executable functions which is available in the form of a Web resource. Each Web service is identified by a name, a unique location (URL) and a set of operations. These operations are specified by their names and input and output types. The information about these operations is available in the form of a syntactic description using a Web Service Description Language. The following describes the three main languages used in specifying services.

- WSDL [35, 44, 150] is an XML-based interface description language for describing network services (e.g., ingoing and outgoing messages, and data types) in the form of the functionality the service offers. Services are represented as network endpoints (or ports) and the network provides a model for representing the communication between multiple services. In addition, WSDL provides a mechanism for Web services to be located (e.g., using a URL), and exchange messages using well-defined protocols. In this model, operations and messages
are described abstractly, and then an endpoint is defined by binding it to a concrete network protocol and a specified message format. The data format specifications for a particular port type along with a concrete protocol constitutes a reusable binding, where the messages and operations are bound to the protocol and the format. A Web Service Description Language is often used in combination with SOAP and an XML Schema to provide Web services over the Internet.

- WS-BPEL [5, 6] is an XML-based executable language designed to enable the coordination and composition of a set of Web services. It is described using Web Service Description Language WSDL and in an XML data format. Web Service Business Process Execution Language is an orchestration language. Thus, it specifies an executable process in which messages are exchanged among different systems via a central controller, where the participants are represented using a state transition model. The Web Service Business Process Execution Language is a behavioural extension of WSDL using a workflow-based approach. It uses control and data flow links to express relationships between multiple invocations. It models the flow of a Web service as a process which is a net-based concurrent description connecting activities that can send (receive) messages to (from) an external Web service provider. A WS-BPEL process can be specified as an abstract process or as an executable one. The main difference is that an executable process can be executed in a standard WS-BPEL engine while an abstract process is limited to specifying the implementation details of a service.
The Web Service Choreography Description Language (WS-CDL) [87] is an XML-based language that models conversations supported by a Web service. The main objective of the WS-CDL is to allow for the specification of “a declarative, XML based language that defines from a global viewpoint the common and complementary observable behaviour specifically, the information exchanges that occur and the jointly agreed ordering rules that need to be satisfied” [87]. In contrast to other standards, WS-CDL describes the behaviour from a global viewpoint rather than from the perspective of one party (choreography). Typically, an interaction starts when one service shows interest in the progress of another service and then establishes a link through some common agreements. Each service is capable of achieving its required functionality in a distributed manner and has a distinct relationship with its peers. In essence, WS-CDL enforces a global observable behaviour of all services in such a way that no single service exercises control over other services.

2.1.5 Semantic Web Service Descriptions

Syntactic-based description of Web services do not have the means to adapt to a changing environment without human intervention. This is because the syntactic specification of Web services provides a rigid set of Web services. Therefore, semantic Web service descriptions were developed to tackle this issue by automating all stages of the Web services life-cycle using explicit or concrete machine-understandable semantics to represent it. Several languages for semantic annotation of Web services
have been proposed. The most widely used include Semantic Markup for Web Services (OWL-S) [9, 105], WSML [48], RDF Schema (RDFS) [163] and WSDL-S [1]. These languages differ with respect to the level of expressiveness and complexity of their construction elements. We discuss OWL-S and WSMF briefly.

- Semantic Markup for Web Services (OWL-S, formerly DAML-S) is one of the most widely used languages that represent Web services with semantic annotation capabilities [9, 105] which defines a specific ontology and associated languages for Semantic Web Services known as OWL. OWL-S uses description logic. Therefore, it supports reasoning about semantic processing such as composition [9]. Semantic Markup for Web Services is made up of three main parts:

  - *The Service Profile* provides the capabilities to express what the service requires (inputs) from and gives to (outputs) users as well as its preconditions and (conditional) effects.

  - *The Service Model* provides a description that specifies how the service works in terms of the control flow and data flow involved in using the service. That is, it models how a client can interact with the service. It provides language constructs for describing a service. These include the sets of outputs, inputs, preconditions and results of the service execution.

  - *The Service Grounding* provides information on how to use the service. It specifies the transport-level messaging information associated with the execution of the actual program that realizes the service.
2.1. SERVICE-ORIENTED ARCHITECTURE

- Web Service Modeling Framework (WSMF) is another initiative to enrich Web services via machine-processable semantics [64]. The WSMF develops and describes Web services and their compositions using a conceptual model. Web Service Modeling Ontology (WSMO) [30] is the foundation upon which WSMF is built. Web Service Modeling Ontology was developed by a group of researchers from the Digital Enterprise Research Institute (DERI) and the Open University (United Kingdom). This language provides a conceptual model and formal constructs, such that the relevant aspect of Web services can be semantically described to facilitate automation of discovering and composition of services. The WSMF is made up of four distinct parts, namely: (i) Ontologies which provide the terminology used by other elements, (ii) Goal repositories that define the problems that should be solved by Web services, (iii) Web services descriptions that define various aspects of a Web service, and (iv) Mediators for resolving interoperability problems.
2.2 State-of-the-Art: Automated Web Service Composition

In this section, we present the state-of-the-art in automated Web service composition. Due to the large number and diversity of existing work on automated composition of service, it is imperative that we classify them into various groups to make the presentation easier to understand. Five distinct groups of approaches are examined based on what mechanism is used to model and generate a composition. Similar groupings have been done in a number of existing surveys on Web service composition [13, 15, 98, 135, 140, 156].

2.2.1 Formal Modeling Languages

Approaches in this category exploit formal modeling languages such as FSM, Petri nets and Unified Modeling Language (UML) to model the problem of service composition. These formal languages provide a means to specify both the composition and its requirements. They also allow easy manipulation of specification using various operation provided by each language, respectively. Formal languages are very useful for consistency checks.

**Finite State Automata Approaches:** One of the earliest works in this category that uses an FSM is by Berardi et al. [19, 21, 22], which presents a formal framework in which execution trees are used to describe the exported behaviour of services (an abstraction for its possible executions). These execution trees are represented using Finite State Machines. In the approach, a service is modeled using two schemata, an external and internal schema which are represented using FSMs.
The external schema specifies the exported behaviour (externally-visible) of services, whereas the internal schema contains information on which service instances execute a given action within the community of services. The approach reduces the problem of composition synthesis into the satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL). That is, both the FSM models of the available services and the target service are encoded into DPDL, and a target service exists if and only if the set of formulas are satisfiable and then an FSM is automatically synthesized. The resulting FSM is further translated into a BPEL process and executed in a BPEL engine. However, this approach does not deal with data and message exchanges during the composition process. An extension of this work uses labeled transition systems to represent services which employ a pre-defined communication topology (a linkage structure expressed as a set of channels) to specify the way services exchange messages [21]. The main aim of this approach is to synthesize a mediator from a given goal service and available service (both specified as labeled transition systems) which orchestrates the execution of the component services such that their conversations are compliant with the goal service specification. Analogous to the former approach, the composition problem is encoded as a Propositional Dynamic Logic (PDL) formula.

A variant of the work by Berardi et al. [23] uses general transition systems instead of FSMs and a composition is performed through the notion of simulation relation [109]. The way the concept of simulation relation is used in this approach differs from how we use it in this dissertation. The approach uses simulation relation as the major technique or means to compute the composition whereas in our case we
2.2. STATE-OF-THE-ART: AUTOMATED WEB SERVICE COMPOSITION

use the notion of simulation relation to help us find a relation between the set of the available services and the composition requirements. In addition, all these works fail to deal with the issue of nondeterminism and controllability during the service composition process.

Pathak et al. [128] propose a Framework for Modeling Service Composition and Execution (MoSCoE) where both the available services and the goal service exhibit infinite-state behaviour. The approach employs Symbolic Transition Systems (STSs) to model services that are associated with guards over infinite domain variables. They use refinement analysis to guide users to refine their composition goal in the case of failure. The framework consists of three steps, abstraction, composition and refinement. Both component services and the goal service are described using UML state machines and are translated into STSs. To this end, the authors apply their composition algorithms to synthesize a composition if it exists. In the case that it does not exist the user reformulates his or her requirements and then tries again. Even so, this approach does not take into account how a specification can be enforced since it does not explore all possible executions of the STSs.

Khoumsi [90] casts the service composition problem as a control problem using a simple input-output automata-based method. The problem is to synthesize an orchestrator Orch from a given set of Web service $S_1, ..., S_n$ and a desired goal $S_0$ such that Orch coordinates the available services to achieve $S_0$. However, this approach is limited to only the input and output parameter description of services and does not capture the behavioural constraints of services.

Jingjing et al. [85] solve the service composition problem using timed automata.
A formal model built on timed automata is used to model Web services and provides an approach for automatic Web service composition. The authors implement an algorithm that automatically generates a timed automaton model for each Web service interface. The generated time automata models are put together by synchronizing them through their branches and end tags. In this case, the equivalent graph is a topology which connects each Web service interface by an equivalence relation. The algorithm implements a composition automation engine which uses the Web service interface description language (WSIL). The Web service interface description language is context-free grammar language developed to describe Web service interfaces. The engine is basically a compiler which takes inputs in the form of WSIL and produces outputs via semantic analysis in the form of a graph or an equivalent tree for a Web service interface. The equivalent tree represents a data structure without loops which can be obtained by performing a breadth-first traversal of the equivalent graph described above. The output is then verified with a verification tool known as UPPAAL. In comparison with this approach, the technique we present in this dissertation does not require an additional verification step to validate the correctness of a composition.

Another approach by Guermouche et al. [70] also proposes a formal technique for composition synthesis and verification which takes into account both quantitative timed properties and qualitative properties (messages, data, and data constraints). The proposed approach generates a mediator which has the ability to enforce time constraints during composition. That is, given a set of timed Web services and a client service, synthesize a timed composition that satisfies the client’s request. In
this composition framework, message sequences are specified using timed conversational protocols where the timed behaviour of a Web service is modeled using a deterministic timed automata based formalism. This specification allows them to use automata clocks to specify the time behaviour such that the transitions of the automata are labeled with timed constraints (guards) and conditions to reset clocks. The algorithm used to synthesize a composition of timed Web services is made up of three phases: (i) creating timed P2P connections between the client service and the discovered services. This stage involves building a timed global automaton representing the timed composition schema; (ii) discovering timed conflicts by generating a mechanism that abstracts implicit timed dependencies from explicit dependencies; and (iii) generating a mediator that tries to establish connections by preventing message mismatches. In the final stage, the authors perform verification of the resulting models using the UPPAAL model checker. Similarly, as we pointed out in the approach by Jingjing et al. [85] above, the technique presented in this thesis does not require a verification step to assure correctness.

**Petri net Based Approaches:** In the context of using Petri nets as a modeling language for automated service composition, Hamadi and Benatallah [71] propose a Petri net-based algebra for modeling Web service control flows which also captures the semantics of a complex service aggregation. The framework provides support for the creation and verification of a composition using various structured-programming constructs (e.g., sequence). The limitation of this work is that it does not explicitly support automatic service composition. Similarly, the literature by Narayanan et al. [116] presents an approach where the semantics of a relevant subset of DAML-S
(now OWL-S) is encoded as first-order logic known as situation calculus. Using this semantics the authors transform the service descriptions into a Petri net formalism and provide decision procedures for Web service simulation, verification and composition. The result of this work has been implemented in a tool which takes as input a DAML-S description of a Web service and synthesizes a Petri net automatically upon which the authors perform various structural analyses.

One of the most recent works using the Petri net formalism is the work described by Xia et al. [175]. The work proposes an approach for reliability analysis of OWL-S processes which employs non-Markovian stochastic Petri-net as the core model. The OWL-S is translated into a non-Markovian stochastic Petri net. A proposed reliability calculation technique which takes as input probabilistic parameters of the service invocations is applied, and a SOAP is used to calculate the reliability of the OWL-S.

In another direction, the work by Cheng et al. [34] employs fuzzy predicate Petri nets to deal with the problem of automatic composition of semantic Web services. A set of Horn clause rules is used to specify a Web service and its fuzzy semantics including the goal service. The authors produce a transformation that maps a service composition task into a Horn clause reasoning problem. The Horn clause set is then transformed into a Fuzzy Predicate Petri Net (FPPN) and a technique known as T-variant analysis [34] is used to check the existence of a composition that realizes a given goal. The authors devise an algorithm which synthesizes a composition based on a given specification as well as an FPPN model that shows the calling order of the selected services.
**UML Based Approaches**  An immense number of research works exist that utilize UML to model the problem of automated service composition. To cite a few instances, the text presented by Wang et al. [165] introduces an approach in which conditional branch structures are used to model the problem of service composition. This approach supports user preferences as well as the ability to adapt to changes in a dynamic real-world environment. In order to model conditional branch structures accurately, the authors employ activity diagrams in UML to represent the dependencies in composite services. They consider two types of user preferences during composition synthesis. The first preference is that the user chooses one class of services over another based on a given constraint. The second type is that the user selects services which offer similar functionality based on a certain priorities assignments.

Similarly, Skogan et al. [152] also propose an approach for semantic Web service composition using Model-Driven Development (MDD). The Unified Modeling Language is used to model Web services. The authors first translate WSDL descriptions into UML models. This translation step allows existing services to be modeled using UML platforms designed for building compositions. Skogan et al. apply MDD techniques to generate a composition based on the UML models of the Web services, which in turn can be translated into executable BPEL specifications. Furthermore, the paper by Skogan et al. presents an open-source implementation that realizes their technique [152]. Given some user preferences and a set of services (each specified by an activity diagram) the authors provide an algorithm that generates all the feasible composite services.
2.2.2 Algebraic and Logic Reasoning

Process algebraic languages such as the Calculus of Communicating Systems (CCS) [110] and the pi-calculus [112] have been recommended as one of the major algebraic methods to model Web service composition [141]. It is no surprise that semantic Web frameworks such as WSMO [30] and SWSF [14] have their origin in process algebra. For instance, Ferrara [65] recommends a process algebra framework as an abstract specification to model Web service. Their framework uses bisimulation analysis to establish whether a service can serve as a substitute for another service in a composition. Moreover, the framework can be used to check for the redundancy of a set of services in a community. Also, because process algebra supports simulation analysis, it allows the use of hierarchical refinement techniques [93] that permit the iterative refinement of an abstract process description into a less abstract one. Furthermore, verification is then carried out to verify various temporal properties. More specifically, the authors use a Language Temporal Ordering Specification (LOTOS) [56] as a process algebra and provide a two-way mapping between WS-BPEL and LOTOS, and general guidelines for translations between WS-BPEL and LOTOS. This approach allows one to verify temporal properties using model checking techniques.

Similarly, the work by Salaïn et al. [146] shows how Process Algebra (PA) can be used to model Web services at the design stage where the system to be developed is represented as an abstract specification using Process Algebra upon which validation can be performed and used as a template for implementation. In addition, the authors show that, by applying reverse engineering, a process algebra specification can be derived from existing Web services interface descriptions. In more detail, the
authors develop a technique to convert BPEL processes into CCS in which various Web service properties specified in temporal logic can be verified and reasoned about. They argue that Process Algebra can be used to tackle various choreography issues. In this work, an equivalence relation is employed to ensure a correct composition, but the composition process is not automatic compared to the work we present in this dissertation.

Rao et al. [139] also present an approach that uses Linear Logic (LL) [69] theorem proving for the automatic composition of semantic Web services. They claim that the semantics of LL provide them with the expressive power to be able to describe both functional and non-functional attributes of Web services. Initially, OWL-S service descriptions are automatically translated into LL axioms. The composite service is automatically generated from the available services using a Linear Logic theorem prover. The target service is expressed as a sequent in Linear Logic and the theorem prover generates all possible compositions which are then translated to process models using a pi-calculus process language. This model can be converted into BPEL workflows models for executions. Generally, the use of the propositional subset of Linear Logic may not be enough to represent composition requirements. Hence, the authors propose to use the principle of partial deduction [101] to allow for more flexibility in specifying users’ goal.

Similarly, Benouaret et al. [17] treat the composition problem as a query one. The problem considered is given a set of candidate compositions which can satisfy the same query, select a composition from the given candidates that satisfies the query. In order solve this problem, the authors employ user preferences to rank the candidate
composition to find the best ones. The technique described by these authors uses fuzzy sets to model user preferences. The next step of the technique attaches the user preferences to the composition query. To this end, an RDF query rewriting algorithm is used to find the candidate compositions that can be used to best answer the query. In addition, a technique is proposed to further generate the best \( k \) data service compositions that satisfy the query. This approach assumes that candidate compositions have already been computed which make this work very different from our work, since the main focus of this thesis is to compute compositions.

It is worth mentioning the work by Ali et al. [3] which provides a formal approach for service composition based on model checking techniques. This approach ensures the correct interaction of the services in a composition by controlling the control flow, and both syntactic and semantic data flow requirements in a single framework. The synthesis of a composition is done using a tableau-based algorithm where Web services are represented as Synchronous Kripke Structures and the control flow specification is specified using Computation Tree Logic. A similar work by Oster et al. [122] also proposes a framework that considers both functional and non-functional properties during service composition based on Goal-Oriented Requirements Engineering, Model Checking, and Qualitative Preference Analysis. Functional requirements are modeled with a Goal Model\(^1\) and non-functional requirements are represented using the nodes of the goal model which specify how satisfaction of each requirement by an existing service contributes to the satisfaction of the non-functional properties.

\(^1\)The notion of the goal model is the same as the one used in the Goal-Oriented Requirements Engineering (GORE) [36] methodology.
A *Conditional Importance Preference Network (CI-net)* [27] is used to specify qualitative properties and trade-offs that may arise during composition. Finally, a model checking technique is devised to generate a composition that realizes a Computation Tree Logic (CTL) specification. In the foregoing discussion, it can be seen that the work by Ali et al. [3] and Oster et al. [122] rely on model checking techniques to promise formal correctness.

Recently, Yang et al. [176] adopt the extended BDI (Belief-Desire-Intention) logic to deal with the problem of Web service composition in the case where a user’s goal is not consistent with the composition goal. The Belief-Desire-Intention model is used to specify a service’s belief, desire and intention, which are mapped into the environment of BDI, the goal of the Web service (and user) and composition schemes respectively. A process model is then used to model the results. In order to allow for a dynamic evolution of their workflow, they use AgentSpeak(L) (a communication language) to express it.

Another recent paper is the work by Belgharbi and Boufaida [16], which introduces an approach that takes context information into account when composing services in a dynamically changing environment. The technique takes a user’s context and the services involved, and then integrates them into both the service description language WSDL and the composition process BPEL. This integration is done at the service description level. The authors describe a method of adaptation of services to the context during the execution of the composition process. Given a context situation, the composition engine analyzes the situation and then generates adaptive services using predefined adaptive rules in the composition engine.
2.2.3 AI Planning

Automated planning is a branch of artificial intelligence that focuses on the automatic generation of plans or strategies that realize a given goal in a given domain. Given a description of the possible initial states of the world and a specification of a set of possible actions, the planning problem entails finding a plan or a strategy to generate a sequence of actions that achieves one of the desired goals (goal states). The AI planning problem can formally be defined as a quintuple $(S, s_0, G, E, \delta)$ such that $S$ denotes the set of all possible states under consideration, $s_0 \subset S$ is the set of initial states of the planner, the set of goal states to be realized by the planner is represented by $G \subset S$, $A$ denotes the set of possible actions and $\delta \subseteq S \times A \times S$ represents a transition relation. In the context of Web services, $A$ is the available services, $G$ represents the target service which specifies the requirements of the client, $s_0 \subseteq S$ is the initial state, and finally $\delta$ is the set of state transitions which denote the precondition and the result of execution of an action. Then in this case, we can adopt existing solutions to AI problems to deal with the problem of automatic service composition. Several approaches have been proposed; we will review them based on the kind of AI technique used.

One of the most commonly used description languages for modeling classical planning problems is Planning Domain Definition Language (PDDL) [59] which has contributed to the development of Web service description languages such as OWL-S. The research by the authors Akkiraju et al. [2], casts the workflow-based service composition problem as a planning domain problem. The authors develop a prototype workflow engine that takes as an input abstract BPEL4WS flow augmented
with semantic annotations in OWL-S and performs runtime discovery, composition, binding and execution of Web services. This engine first translates OWL-S service descriptions to PDDL; the output from this step is then passed into IBM’s Planner4J planning framework for further analysis. The planner module requests a concrete service for each task in the abstract BPEL flow at runtime. If this process fails the planner uses the available services to solve the planning problem. However, since the creation of the abstract flow is manual, this approach is considered a semi-automatic approach.

In a similar fashion, Zeng et al. [179] also formulate the service composition problem as a goal-directed planning problem which utilizes a comprehensive rule inference mechanism to dynamically generate a composition schema. The approach takes in three inputs: (i) a set of domain specific service composition rules (business rules) which have associated pre-conditions, and obligations; (ii) a description of the user’s business objective or goal; and (iii) a description of business assumptions (organizational rules and structures). These concepts are then modeled based on a consistent service ontology which the authors have designed. To that end, the authors propose a three-phase rule inference mechanism which generates a composition service schema incrementally. The first phase known as Backward-Chaining is to determine all the possible actions (from the business objective to the initial state) that must be executed in order to satisfy the business objective. The second step involves using a technique called Forward-Chaining inference that attempts to take the result of executing tasks in the previous phase and then determining which tasks may need to be included. The final step provides the Data Flow inference.
2.2. STATE-OF-THE-ART: AUTOMATED WEB SERVICE COMPOSITION

The work by Rao et al. [138], presents a mixed initiative framework for semantic Web service discovery and composition which does not attempt to automate all decisions, but assumes that the users should retain close control over many decisions while having the ability to selectively delegate tedious aspects of their tasks. They used an AI planning algorithm known as GraphPlan to build their composition engine which combines rule-based reasoning on OWL ontologies with Jess (a rule engine and scripting environment for Java platforms) and planning functionalities. The main use of planning here is to provide suggestions of composition schemas to the user, instead of enforcing decisions which form the ultimate goal of this work.

In much the same way, Wu et al. [174] employed graph-based planning to solve the service composition problem. The approach considers both process heterogeneity and data heterogeneity problems. They implemented their own definition of an abstract semantic Web service built on top of SAWSDL and WSDL-S. Then, the authors extended GraphPlan that automatically generates the control flow of a Web process. The system automatically generates an executable BPEL process from a given specification of the initial state, the goal state and a semantically annotated Web service description in SAWSDL. Data mediation is done using assignment activities in BPEL or by a data mediator which may be embedded in a middleware. At runtime the data mediator converts the available service into the format of the input message of the operation which is invoked when called by the BPEL process.

Sirin et al. [151] attempt to leverage the Hierarchical Task Network (HTN) planning technique for the automated composition of semantic Web services. The authors
are motivated to use this technique based on the fact that the concept of task decomposition in HTN planning is very similar to the concept of composite process decomposition in OWL-S process ontology. They build a system that translates OWL-S service descriptions into SHOP2 (a domain-independent HTN planning system for HTN) [118] and then the authors provide a method to automatically synthesize a feasible composition plan. The system is also capable of executing information-providing Web Services during the planning process. They prove the correctness of their approach by showing the correspondence to the situational calculus semantics of OWL-S.

Peer [129] shows how the Partial Order Planner known as Versatile Heuristic Partial Order Planner (VHPOP) can be combined with a replanning algorithm for automatic service composition. The author provides a definition of semantic Web services which is then translated into PDDL as an input for VHPOP. The PDDL description of the Web service is fed into VHPOP as well as a set of links between tasks to avoid. One or more plans are automatically generated which may be partially defined. During runtime, execution is done one step at a time since the generated plan(s) does not necessarily ensure correct execution. Hence, if a plan fails, replanning is performed and a new plan is produced, given the conditions of the failure; however, if the execution of a plan is successful, there is no need to replan and one can move on to the next task.

Klusch and Gerber [92] present an approach similar to that of Peer’s. However, the authors built their framework on OWL-S descriptions rather than develop their own ontology service language. Similarly, the OWL-S descriptions are converted
into PDDL descriptions and then fed into their AI planner. They used a hybrid AI planner known as Xplan which combines the benefits of both graph based planning and HTN. Their approach increases planning efficiency in two ways. The graph-plan based FastForward-planner always finds a composition if it exists in the action state space, whereas HTN planning provides decomposition planning techniques. In addition, their planner supports replanning components which automatically update or react to changes during the composition planning process.

Recently, Sohrabi and McIlraith [154] presented an approach that supports customization, optimization and regulation enforcement during composition construction time by incorporating preferences and regulations into HTN planning. This work builds on the technique presented by McIlraith and Son [106] by extending and customizing Golog [66] to support personalized constraints and nondeterminism in sequential executions and then the authors redesigned ConGolog, the interpreter of Golog to take care of these changes. Interestingly, this development took place alongside the development of the definition of OWL-S and was one of the first works to use Semantic Web Services as an input to planners through translation to PDDL.

One of the most popular and interesting research contributions employing AI planning to tackle the problem of automated service composition in a controller synthesis settings is the work by Pistore et al. [24,88,133], which presents a model checking based planning approach that uses transition systems to model Web services that communicate via exchanging messages. The authors adapt symbolic model checking techniques to planning in order to deal effectively with nondeterminism, partial
observability, and complex goals. They use this technique to find a parallel composition of all the available services and then synthesize a controller that ensures that the composed service satisfies the given requirement by controlling it. That is, given a set of available services $W = W_1, W_2, \ldots, W_n$ and $\rho$ describing the goal specification, the authors compute a controller (plan) $W_c$ such that $W_c \triangleright (W_1 \parallel W_2 \parallel \ldots \parallel W_n) \models \rho$, where $\parallel$ is the composition operator. Their work converts OWL-S processes to state transition systems and then goals are expressed using a requirement specification language called EAGLE. Both the state transition systems and the goals are fed into an MBP planner. Even though this approach can produce correct plans, it however suffers from scalability problems partly due to the way goals are expressed. Pistore et al. [31, 75, 132, 136] tried to solve the scalability issues in the approach presented above by the definition of an appropriate model for providing a knowledge level description of the component services which uses BPEL workflows instead of OWL-S process models. The work has been incorporated into a well-known automated service composition project called the ASTRO framework [103]. The main difference between the work by Pistore et al. and our work is that Pistore et al. make use of AI techniques in generating a controller while in our work we make use of supervisory control theory. In addition, our approach makes use of Labelled Transition Systems augmented with guards and variables while the approach by Pistore et al. make use of State Transition Systems.

Continuing with the AI approaches, Zou et al. [183] use numerical temporal planning to tackle the problem of dynamic Web service composition which considers quality of service properties. One unique feature of this approach is that it does
not rely on existing predefined workflows but it automatically generates temporal and numeral specifications from a QoS-aware composition task. This approach is basically made up of two steps. Firstly, a quality of service aware composition task is translated into a PDDL formula which is further transformed into a cost-sensitive temporally-expressive planning problem. This stage characterize the service composition problem as a numeral planning problem involving time and cost optimization. Finally, the temporally expressive planning problem is solved using a SAT-based cost planning solver developed by the group. This solver deals with logical reasoning, temporal planning and optimization of QoS composition.

Other recent research such as that of Khanfir et al. [89] propose a technique called intentional oriented architecture which combines a user’s query and the functionalities of a set of services to express the composition goals. The objective of the approach by Khanfir et al. is to satisfy a user’s requirements while taking into account a user’s request (expressed by the user in a natural language that specifies the user intentions), context and a service quality property. The authors present a framework called a quality context adaptable intentional service framework (QC-AISF). The framework is divided into two modules. The first one is a publishing module that allows the publication of adaptable intentional Web service and the second module is a composition and selection module that enables the composition and selection of adaptable intentional Web services that fulfill a user’s intention, his or her context and preferences for quality of service. According to the authors of this paper, the framework has been implemented in a tool using a modified version of OWL-S.
Silva et al. [45] present a Genetic Programming (GP) technique that simultaneously considers functional requirements, user constraints and quality of services in the composition process. The approach uses GP to evolve solutions based on their functional correctness and their overall quality of service. A tree is used to represent a possible solution for a composition, such that the non-terminal nodes are used to specify the composition flow, and the terminals are used to specify atomic Web services. A population initialization algorithm is developed to generate a composition. However, the technique does not show how multiple branching conditions could be handled. Moreover, the tree representation used in this approach may result in a redundant situation in which the same service appears multiple times within a candidate tree, which limits the ability of this approach to guarantee functional correctness. To solve this problem the authors present another work [46] which represents the solutions as Directed Acyclic Graphs.

In another direction, the work by Markou and Refanidis [104] employs contingent planning to model the problem of automated semantic Web service composition. The research considers Web services specified in terms of their preconditions and effects over ontological concepts. In addition, the Web services are allowed to have several alternative outcomes, such that each outcome is marked with its probability of occurrence. The approach starts by representing a Web service domain as a probabilistic planning problem. Next, a technique is used to determinize the model as well as keeping records of information relating to the cost and probability of the original model. Then a deterministic planner is used to solve the problem repeatedly and stops when the limit specified by the user is reached or there are no more plans...
to generate. In the end, the generated deterministic plans are merged into a decision tree using a planner know as MAPPPA. This technique has been integrated into a tool call MADSWAN. Despite the result presented by this work, the technique does not show how alternative plans can correctly be generated.

Other recent works using AI planning to deal with Web service composition can be found in the literature [63, 115, 119, 155]. However, most of the AI planning techniques assume that the behaviour of services is deterministic and, hence, these approaches fail when unexpected events occur [13].

2.2.4 Workflows

One of the most recent contributions based on workflow techniques is the work by Paik et al. [124], which presents a workflow orchestration approach to allow for nested multilevel composition for achieving scalability. The work proposes four stages for performing automatic service composition. This approach deals with situations where composition goals cannot be achieved in only a one-step process. The first stage involves planning a workflow of individual service types using a hierarchical task network planner. The second step provides a mechanism for locating services from a service repository. The next stage involves the deployment and execution of the most suitable candidate services which are selected based on their non-functional properties. The final stage involves the execution of services; in the case where there is a failure, the execution stops and replanning is performed again. The above stages are used to translate a service from an abstract workflow to a concrete one. This approach tries to deal with both functional and non-functional properties during
service composition. The implementation consists of a top-level architecture which is made up of the following: (i) a *WorkflowGenerator* that creates an abstract workflow that satisfies a functional specification of a user’s request, (ii) a *ServiceDiscoverer* class that provides a means for service discovery, (iii) a *ServiceSelector component* which selects service instances based on non-functional properties, and (iv) a *Service selector* which executes the service selected in the previous stage.

The text by Majithia et al. [102] presents a framework for automated composition of workflow with the help of semantic Web technologies. The framework consists of an *Abstract WorkFlow Composer* which accepts as input a user-specified high-level objective and transforms it into an abstract workflow. This transformation is done by querying an existing workflow repository to ascertain if the same request has been processed previously. Otherwise, it will backtrack to find a similar set of services that provides the same functionality. A *Concrete Workflow Composer* takes the generated abstract workflow as input and tries to match the individual tasks with available instances of actually deployed services using a matchmaking algorithm. If the matching process is successful, the matchmaking algorithm generates an executable graph which serves as an input to a *Workflow Manager Service* which combines the available services based on user-specified preference. If the matching process fails, the *Abstract WorkFlow Composer* is recursively called to find a combination of services that can provide the required functionality.
2.2.5 Graphs

Graph theory and algorithms utilizing graphs as a means of representation have been widely used to model the problem of service composition. To model Web services and their compositions, graphs are employed to represent and model the structure, data flow and dependencies as well as to specify composition requirements of a given composition task.

A recent paper by Upadhyaya et al. [160] presents an approach for service composition based on mining process knowledge from the Web. Based on human written instructions in the form of web pages, their approach is able to guide the client to perform an action by discovering and integrating multiple services. A mining technique is used to construct a knowledge base for a process which exploits the structure of the *how-to descriptions*. This technique is capable of identifying a task model from a written instruction in the Web pages and combining various services to execute a process. The first step of this approach allows users to select one of the several *how-to instruction*\(^2\) Web pages that are relevant to their desired goal. Then, a search is performed to find a match among the Web pages that satisfy the user’s scenario. A task model\(^3\) is then extracted from these selected Web pages and then data and control flow requirements are identified. The authors consider the task relationships from a given set of task models as the control flow requirements and the data-dependencies between services serve as the data flow requirements. Next, a composite service is produced from the extracted models. The approach also automatically generates a

\(^{2}\text{How-to instructions in the Web are a knowledge derived from human activities [160].}\)
\(^{3}\text{A task model is a logical task that have to be carried out in a process to reach a user’s goals [160].}\)
UI (User Interface) to execute a task.

Another related work in this category is the work by Yong and Yang [177], which focuses on providing a genetic algorithm approach to semantic-based automatic service compositions with functional and non-functional requirements at design time. This approach is made up of three design stages: (1) the first stage synthesizes a composite Web service from a set of non-executable abstract workflows. To do the synthesis, both functional and non-functional requirements are specified and fed into a service integrator which manually or semi-automatically generates a workflow (i.e., an abstract plan) that satisfies the given requirement; (2) in the second stage, each abstract activity is represented with a relevant candidate service through discovery, which involves functional matchmaking between concrete services and abstract activities; and (3) finally, the fittest service is selected for each abstract activity from the set of all candidate services discovered in the previous stage. This selection must meet certain specified conditions such as QoS. The authors propose to find a more efficient fitness function during this process.

The paper by Hashemian and Mavaddat [72] presents an approach for automatic Web service composition by combining techniques based on interface automata and graph theory. Particularly, the authors use interface automata to model Web services. The interface automaton is used to model the inputs and outputs of a component along with the temporal ordering of the actions it performs. A dependency graph is used to model the data dependencies between the components services, such that the nodes of the graph correspond to the inputs and outputs of the Web services.

---

4 “A state-based model, similar to finite state diagrams, for representing behaviour of software components” [72].
and the edges represent the associated Web services. The algorithm for synthesizing a composition is made up of two steps. The first step is to find a set of Web services that can potentially participate in the composition and the second step is to generate a dependency graph based on a given user request using the set of Web service found in the previous stage.

A related approach presented by Li et al. [99] proposes a graph base algorithm to model the problem of automated Web service composition. The algorithm developed by these authors initially generates all possible service combinations beforehand and then stores them in a relational database system. Now, when a client presents a request, an algorithm developed by the authors is used to compose a set of SQL queries to search in the database and return the best solution with regards to quality of service properties. This approach does not show how functional correctness can be achieved.

The paper by Ba [8] describes an automatic service composition approach that uses a query rewriting technique. The approach takes an abstract specification of a composition and a definition of a set of concrete services as inputs. Given these higher-level specifications, the author of this article uses an improved version of an algorithm known as MiniCon query rewriting that computes a MiniCon Description (MCD). Next a bipartite graph is used to reduce the rewriting problem to an exact cover problem. A technique is then used to compute a composition from this graph.


Discussion: Existing Approaches Overview

The literature review presented above highlights the different flavours of automated service composition approaches that already exist. The literature reveals some key characteristics of automated service composition approaches. These include the level of automation of a given approach and whether or not it considers non-functional requirements (i.e., quality of services properties) during composition. We also investigated approaches that explicitly support data and control flow. Some of the approaches presented above address the issue of nondeterminism and partial observability but none of these approaches treats the issue of controllability as presented in this dissertation.

The problem of automated service composition has been tackled using AI planning techniques more than any other techniques, but not every problem can be modeled by planning. For instance, most of the AI planning techniques assume that the behaviour of services is deterministic and hence, these approaches fail when unexpected events occur. Moreover, they tend to neglect the context in which composition synthesis takes place by anticipating in advance how services will interact. Also, if plans generated by AI algorithms are not able to achieve a given target goal, the only way to resolve this issue is replanning which might deliver a result too late. Hence, planning may not be the best option to deal with the problem of service composition in this case. Moreover, classical AI planning techniques can generate only linear sequences of actions and the outcome of each action must be known in advance.
With regards to the issue of correctness which is the main focus of this dissertation, most of the approaches we have discussed so far do not explicitly treat it at all or deal with it as a second step after composition by relying on verification and model checking techniques. Most of these approaches lack formal assurance and the concept of correct-by-construction. Hence, they fail to provide composition algorithms that are complete (i.e., able to find a possible composition whenever it exists) and sound (i.e., can deduce that the composition generated achieves the goal service).

2.3 Discrete-Event Systems

In this section, we give a brief overview of the supervisory control theory of DES [33, 137, 172]. We start with a brief summary of the notion of formal languages.

2.3.1 Preliminaries

We define $\Sigma^*$ as the set of all possible strings over some finite alphabet $\Sigma$, including the empty string $\epsilon$ of length zero. A language $L$ over $\Sigma$ is any subset of $\Sigma^*$. In particular $\{\epsilon\}$, $\Sigma$ and $\Sigma^*$ are languages. Let $s \in \Sigma^*$, a string $t \in \Sigma^*$ is a prefix of $s$ if $s = tu$ for some $u \in \Sigma^*$ (where $tu$ is the concatenation of the string $t$ and $u$). The prefix-closure of a language $L \subseteq \Sigma^*$, denoted by $\overline{L}$, consist of all the prefixes of all strings in $L$, i.e., $\overline{L} = \{s \in \Sigma^*|\exists t \in \Sigma^*, st \in L\}$. A language $L$ is said to be prefixed-closed if $L = \overline{L}$.

A DES is usually modeled by a finite automaton defined as $G = (\Sigma, Q, q_0, \delta, Q_m)$, where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $\delta$ is the transition function (partial) such that $\delta : \Sigma \times Q \to Q$, and $Q_m \subseteq Q$ is the set of marked
2.3. DISCRETE-EVENT SYSTEMS

state. In DES $\Sigma$ is the finite alphabet of events. We extend the transition function $\delta$ to strings (words) by defining it inductively as follows: $\delta : \Sigma^* \times Q \to Q$ is such that $\delta(\epsilon, q) = q$, $\delta(s\sigma, q) = \delta(\sigma, \delta(s, q))$, with $\sigma \in \Sigma$ and $s \in \Sigma^*$. The language generated by $G$ is $L(G) = \{s \in \Sigma^* | \delta(s, q_0)\text{is defined}\}$. This represents all the possible event sequences that some process $G$ can go through (i.e., the sequences need not terminate at marked states). On the other hand, the marked language of $G$ is $L_m(G) = \{s \in L(G) | \delta(s, q_0) \in Q_m\}$. The marked language describes the subset of event sequences that represents the completion of some tasks. For more details, we can refer to the text by Cassandras and Lafortune [33].

2.3.2 Supervisory Control Theory of DES

As introduced previously in Section 1.1.3, the standard DES originated with Ramadge and Wonham in the 1980s [137]. Although there are many variations and embellishments of SCT, the standard Ramadge–Wonham Theory [137, 172] considers a DES modeled at an untimed level of abstraction which relies on feedback control to restrict the system to achieve a given set of specifications. In this framework, the process (system) to be modeled which is characterized by sequence of actions or events is assumed to be behaving undesirably in some way; such a process called the plant $G$ is modeled with an FSM as defined above (Section 2.3.1). Events are represented by the transitions in $G$, and the language generated $L(G)$ represents the behaviour of $G$. The language $L(G)$ contains strings that may be unacceptable because they violate some rules or nonblocking conditions that we wish to impose on the system. The undesirable behaviour could be states where $G$ blocks, through a
deadlock or a livelock, or inadmissible states. To this end, the behaviour of $G$ is not satisfactory and must be controlled by restricting the behaviour to a subset of $\mathcal{L}(G)$. A supervisor $S$ is introduced in order to restrict the behaviour of $G$.

In the basic model, the event set $\Sigma$ of $G$ is partitioned into two disjoint sets, namely, the set of controllable events $\Sigma_c$, meaning an event in $\Sigma$ that can be disabled and the set of uncontrollable events $\Sigma_{uc}$, meaning an event in $\Sigma$ that cannot or should not be prevented from occurring. According to Cassandras and Lafortune [33], an event might be modeled as uncontrollable because it is inherently unpreventable or it models a change of sensor readings not due to a command; it cannot be prevented due to hardware or actuation limitations; or it is modeled as uncontrollable by choice, as an example when the event has high priority and thus should not be disabled, or when the event represents the tick of a clock. Ideally, a supervisor is given by a function $S : \mathcal{L}(G) \rightarrow 2^\Sigma$ that maps the sequence of generated events to a subset of controllable events to be disabled. The synchronous product of $S$ and $G$ is the marked language denoted by $\mathcal{L}_m(S/G)$ that represents the behaviour of the plant $G$ under supervision of $S$. The work by Ramadge et al. [172] states the necessary and sufficient conditions for the existence of a supervisor. In this framework, a specification given as an FSM provides the desired behaviour of the plant, and is called the legal language $\mathcal{E}$. A plant $G$ is called controllable with respect to a specification $\mathcal{E}$ if for any string $s$ from the prefix closure of $\mathcal{E}$, there are no uncontrollable events $e$ that could be generated by $G$ at the state reached by $s$, such that $se$ would not be in the prefix closure of $\mathcal{E}$. That is, if something cannot be prevented, it must be legal. The Supervisory Control Architecture addressed by this work assumes that the plant spontaneously generates
all events and the role of the supervisor is to enable/disable controllable events (since $S$ is not allowed to ever disable an uncontrollable event). The supervisor $S$ guarantees not only deadlock-freedom (nonblocking) and adherence to the specification $E$, but also minimal restrictiveness. These strong theoretical formal promises set DES apart from other approaches like AI planning [88, 183].

The complexity of Ramadge and Wonham’s model of supervisors’ synthesis is polynomial in the number of states of the plant and the specification which presents a limitation with respect to its application due to the state explosion problem. Thus, several extensions have been proposed. One of these extensions is modular supervisory control [173] which allows for multiple specifications and multiple supervisors to enact control on a single plant independently. Nonblocking of the composite system is achieved only under certain conditions. The paper by Lin and Wonham [100] also introduces the notion of partial observation in which the supervisors are no longer assumed to observe all events that the plant generates but instead must base their control decisions on only partial information. Another extension of the standard DES is hierarchical DES [180] in which a two-level hierarchy of control is considered. The high-level system is an aggregated model of the low-level process and impacts control through an information channel. In the decentralized control approach proposed by [145], distributed supervisors are considered, each of which only has partial observations and partial control, and each performs separate actions so that together some overall goal is achieved. Other relevant extensions include real-time constraints [29] and probabilities [96, 125, 159]. The Ramadge and Wonham’s model forms the basis on which we shall build our framework.
As mentioned in Section 1.1.3, Supervisory Control Synthesis has several benefits which are as follows. It results in a correct-by-construction control synthesis, and the generated controller is minimally restrictive by preventing a system behaviour only if it violates the system requirements. It also relies on automata theory to provide a well-defined syntax and semantics for modeling systems which could be very useful for specifying services. In addition, DES provides a standard way that can be used to model various business logics and requirements in a dynamic environment.

2.3.3 Application: Discrete-Event Systems and Software Modeling and Automation

This is not the only work proposing to use DES to simplify and automate the development of software systems. Discrete–event systems theory has been applied to various domains in software engineering and related technologies such as concurrency control, communication systems, software failure diagnosis, database transactions analysis, protocol verification, feature interaction and execution of workflows and some of them have produced promising results [52]. In this section, we shall review some of these works.

In the area of operating systems (OS), Phoha et al. [131] present an approach to control software systems based on Supervisory Control Theory. Supervisory Control Theory is used to model the execution process of a software application such that the actions of the operating system are restricted with minimal modification of the underlying operating system. The proposed software management technique includes (i) a fault mitigation technique in software systems; (ii) real-time control of
2.3. DISCRETE-EVENT SYSTEMS

software systems; (iii) runtime behavioural modification and control of the OS with insignificant changes in the underlying OS; (iv) modeling of the OS-Application interactions as symbols (events) in the formal language setting; and (v) accommodation of multiple control policies by varying the state transitions.

Another work by Santos et al. [148] presents a supervisory control service architecture to support end users of flexible Process Aware Information Systems (PAISs) when executing processes. The authors implement the notion of controllability in such a way that at any point in time during a process execution the supervisor provides a list of possible next steps the user can take by disabling or enabling events.

Karsai et al. [86] present a paradigm for modeling self-adaptive software systems based on Supervisory Control Theory. Specifically, the authors apply hierarchical control systems theory of DES to deal with the problem of adaptation. In this work, adaptation is represented as a higher-level system function that performs optimizations, manages faults as well as direct the execution of a system towards its goal. Similarly, Côté et al. [43] employ hierarchical control systems theory of DES as a theoretical foundation to address control issues in component-based software development. The main emphasis of this work is to enforce control in component-based systems that guarantee safety, deadlock-freedom and a correct-by-construction system. In their implementation, components are Web services implemented as BPEL processes and then translated into automata. Next, the authors propose an algorithm to compose and enforce control on this process based on a hierarchical control variant. The main result of this paper shows how to enforce control on component-based software systems.
In the domain of workflow execution, the work presented by Jensen et al. [83] adopts SCT techniques for specifying and synthesizing scheduling controllers in the workflow management paradigm. They use finite state automata to specify tasks and their dependencies, and devise a technique to check the existence of a scheduler and construct it algorithmically. The generated scheduling controller of the workflow system ensures correct and secure executions of tasks. In a similar manner, the research work by Wang et al. [167] describes a technique that uses DES for the safe execution of IT automation workflows. The technique generates a supervisor that ensures that the execution BPEL workflow does not result in a deadlock or lead to any “forbidden” states given by the specification.

Thistle et al. [158] employ modular supervisory control and the notion of partial observation to deal with the problem of feature interaction in telecommunication services. Finite state machines are used to model services and features for multiple subscribers in a telephone network. Modular supervisors are used to detect inconsistent interactions between features.

Supervisory control theory presents certain concepts which are useful in the domain of database system analysis. For example, there exists work that uses standard DES for performing and comparing different database transactions, and execution protocols [95].

There have been some attempts to use DES for modeling protocol verification. Rudie and Wonham [143, 144] apply decentralized DES control to model agents in telecommunication networks.

Discrete–event systems have also been applied in the area of concurrency control.
and have been very promising. For instance, Dragert et al. [53] applied classical Ramadge and Wonham control theory to synthesize a controller $S$ for concurrent Java code $G$ [53]. The number of threads does not change during runtime (static threads). Given Java code devoid of concurrency control code, certain portions are marked as specification-relevant events in the code. These events are controllable. The system specifications are modeled using FSA. Based on supervisory control theory, a supervisor is automatically generated which is used to inject concurrency control code into the original code. Auer et al. [7] extended this work to handle dynamically instantiated and terminated threads using a Petri net formalism and an online limited-lookahead state-space search technique. Yet another take on this work is the Gadara project by Lafortune et al. [166,168] which uses a variant of DES called supervision based on place invariant (SBPI) to generate controllers that guarantee deadlock freedom for concurrent code. The Gadara project employs a special class of Petri nets known as Gadara nets to systematically model multithreaded programs with lock allocations and release operations. Siphon analysis is used to identify deadlocks in a C program and then SBPI is used to ensure that these deadlocks are avoided during execution. A similar approach by Iordache and Antsaklis [82] led to the development of a software tool known as a concurrency tool suite (ACTS) which takes a specification in a form of high-level specification (HLS). The plant model of the concurrency process and a specification are extracted from the HLS for supervisory control. The supervisory control synthesis is applied to generate a supervisor [82].
From the foregoing discussion, we can infer that DES is increasingly being ap-
plied to tackle various problems in software engineering. This increase is due to
the several benefits that DES provides and the fact that it is a well-developed body
of knowledge. It provides several applications in the software engineering domain.
Table 2.2 summarizes the growing application of DES to software engineering prob-
lems. This results further motivates our application of DES to the problem of service
composition in this dissertation.

2.4 Most Closely Related Work in DES

Having presented the state-of-the-art approaches to service composition (Section 2.2)
and a literature review on DES (Section 2.3.3), we now present the most closely
related work to this dissertation, which lies between automated service composition
and DES. In fact, to the best of our knowledge there is no existing literature that
addresses the problem of service composition using DES as we have proposed. That
is, despite a lot of work on service composition (also in a formal sense) there is no
existing work that has been formulated as presented in this dissertation. In spite of
this limited number of closely related work, the objective of this section is to present
the limited number of closely related papers, and provide a clear understanding of
the problem area and a view of where the contributions of this dissertation work fall
into.

Wang et al. [167] apply discrete-event control theory to IT automation workflows.
Their technique generates a supervisor which detects flaws in workflows in a way that
is similar to static analysis. In effect, it allows safe execution of flawed workflows by
2.4. MOST CLOSELY RELATED WORK IN DES

Table 2.2: Application of Discrete-Event Systems to Software Engineering (the direction of the arrow indicates the progressive use of DES to tackle software engineering problems)

dynamically avoiding runtime failures. First of all, the authors translate workflow languages such as BPEL into automata that model the control flow and reachable state space. A technique is used to identify uncontrollable transitions. The automata
models are further translated into Petri nets. A control synthesis algorithm is used to enforce constraints during execution. More recent work by Wang and Nazeem [169] investigates the use of supervisory control in the artifact-centric design paradigm. They present a framework to synthesize an artifact-centric process automatically from a given set of artifacts and services such that a correct execution is guaranteed by properly handling uncontrollable events. The synthesis problem in the artifact-centric domain is analogous to the composition problem in SOA. However, the focus of their paper is not on automatic composition as considered here.

Balbiani et al. [12] apply a variant of supervisory control theory in which the system requirements are specified in modal logic to model an abstract form of service composition where nondeterministic communicating automata are used to represent Web services. The composition synthesis problem considered here is, given a community of services and a goal service, to synthesize a mediator such that the triplet client/mediator/community is equivalent to the goal service; the equivalence is given by a bisimulation relation modulo some hidden internal actions and communications. The authors conclude that automatic synthesis is decidable and prove that the problem reduces to a control problem. The composition problem considered in this paper is restricted to synthesizing a specification (mediator) that realizes a given goal but does not show how to actually orchestrate the services in terms of data and control flow requirements during execution time. However, in our case we shall consider both cases. Additionally, the control problem the authors employ differs from ours. The version of controller synthesis used in their work is the following: given an automaton and a logical formula, find an automaton called the controller satisfying some
controllability and observability constraints and such that the synchronous product between the automaton and the controller satisfies the formula. Also, unlike our work, the paper approaches the composition problem from a completely theoretical viewpoint without practical or experimental validation of their claims. Finally, the way we propose to model services is totally different from what is considered in the paper by Balbiani et al. For instance, the proposed approach considers data and guards when specifying services along with nondeterminism issues. In related work, Darondeau et al. [47] also use the supervisory control theory of Ramadge and Wonham in the context of the Service-Oriented Architecture to enforce a modal specification on a service provider modeled by a finite LTS. However, the research treated in the work by Darondeau et al. is not about service composition but mainly about service descriptions.

Based on the work presented, we can infer a number of key ideas that connect the problem of automatic service composition to supervisor synthesis. That is, the problem of synthesizing a composition can be reduced to the problem of supervisor synthesis when available services and a goal specification represent the plant and the legal language (desired behaviour), respectively, in DES. In this way, the problem of orchestrating data and control flow requirements can be achieved by using the notion of controllability in DES where the supervisor enacts control by disabling and enabling certain actions in order to enforce the given goal. Hence, we can apply existing DES techniques and algorithms to address the problem of automatic service composition.
2.4. MOST CLOSELY RELATED WORK IN DES

2.4.1 Comparison with the State-of-the-Art

In this section, we present a comparison between the work in this dissertation with the existing literature.

The approaches presented in Section 2.2 provide several techniques to deal with the problem of automated Web service composition, but are completely different from the work we propose in this dissertation. Nevertheless, they provide insight and demonstrate various interests in the topic. The synthesis problem considered in AI planning is similar to the supervisor synthesis in DES. However, while DES provides a correct-by-construction approach to synthesize controllers, AI planning involves the generation of plans by planning and replanning without any mechanism to prevent failures before they occur. That is, the use of supervisory control theory ensures that all illegal behaviours of a system are prevented before they can occur, whereas the use of AI planning will only generate plans without taking into account failures that may occur during execution. The only way to deal with failures when a plan has already been generated is to generate an alternate plan again. Most of the AI planning techniques assume that the behaviour of services is deterministic hence, these approaches fail when unexpected behaviours occur [13]. The generation of plan(s) does not necessarily mean a correct execution is guaranteed at runtime.

The work by Pistore et al. [24,88,133] presents an approach similar to the proposed work but relies on AI planning techniques to synthesize controllers. The approach captures the notion of nondeterminism in services. This approach differs from ours in that it uses AI techniques whereas the proposed work is built on DES methods. In fact, the proposed approach deals with the notion of controllability in
which set of actions is partitioned into controllable and uncontrollable. This notion set our approach apart from all other approaches.

With respect to control in service composition, Khoumsi [90] treats the problem of service composition as a control problem. As mentioned in Section 2.2.1, this approach is only limited to the input and output descriptions of the behaviour of services (behavioural constraints) and does not capture sufficiently what a service is in the real world. This approach is not based on any control theoretic technique compared to DES. Ali et al. [3] also present a formal framework analogous to the proposed work but their technique relies on model checking techniques for composition synthesis. Similarly, the work by Oster et al. [122] also uses model checking techniques but it does not consider any form of nondeterministic description of services which we take into account in our framework.

With regards to cases where a composition cannot be realized from a given set of available services due to incomplete specification, the approach proposed by Pathak et al. [128] provides a means to achieve this through reformulation of the specification. The way their approach achieves this construction is to recursively refine the goal service until a composition can be found. However, in our case, we reconstruct the specification using the concept of a simulation relation between the specification and the given set of available services.

In terms of optimal composition for a given set of services and a specification, none of the approaches presented above addresses this issue. However, this dissertation tackles this issue by ensuring that the controller is minimally restrictive.

In a nutshell, relative to supervisory control theory, none of the above approaches
(Section 2.2) is able to prevent a system from violating its requirements or guarantees that the system requirements are always satisfied in the presence of failure.
Chapter 3

Supervisor Aware Service Composition Architecture (SASCA)

This chapter reports some of the major contributions of this dissertation. It presents (a) the description and the specification of services in the framework proposed by this dissertation, (b) the new formal service composition framework, (c) a set of algorithms to synthesize a composition based on the formalism introduced in (b), and (d) the proofs of correctness and completeness of the approach as well as a brief discussion of the computational complexity of the algorithms.

3.1 The Service Composition Problem by an Example

In this section, we present the composition problem using another variant of Example 1.1.1. The example described here is a modified version of a well-known example for illustrating Web service composition in the business domain. As mentioned previously (Section 1), the service composition problem comes into play when a request
to a service (component service) is not satisfied by a single service and then multiple services are identified and constructed into a new service (composite service) to satisfy the request or the desired functionality through orchestrating the services involved.

Let us consider a Flight Reservation and Purchase System (FRPS) that offers customers travel packages by allowing customers to make a reservation for a specified airline and to make payment in order to reserve the flight. All interactions are managed by Web services. The objective of the Flight Reservation and Purchase System can be attained by composing an Airline Service, a Bank Service, a Hotel Service and an On-line Customer Interface Service. We assume that these services are represented in WS-BPEL. The main challenge is how to compose these services so that the user can directly ask the combined service to reserve and purchase a ticket satisfying some given system requirements. In the following, we will provide an informal description of these services.

- **Airline Service**: The Flight service is designed to receive requests for booking a specified flight for a given date and location. It checks an internal database for flight availability, and sends an offer with a cost and a flight schedule in response to the client’s request. The client can either accept or refuse the offer; if the client decides to accept the offer, the FRPS will book the flight and provide additional information such as an electronic ticket.

- **Bank Service**: The Bank service is designed to receive a request to check that a credit card, debit card or money order can be used to make a payment and provides an option for its clients to check their current balance or withdraw
from the account to make a payment for a purchase. The transaction may fail if the card provided is not valid or if there are not sufficient funds in the client’s accounts.

- Hotel Service: The Hotel service accepts requests for providing information on available hotels for a given date and a given location. It checks for the availability of hotels and selects a specific hotel based on the client’s request and returns an offer with a cost and other hotel information. The external service that invoked the Hotel service can choose to refuse or accept the offer. In case of acceptance, the hotel proceeds with the booking and sends a confirmation message to the client.

- On-line User Interface Service: This service serves as a customer interface through which the client can interact with the Flight Reservation and Purchase System. It receives input messages from the user and in return sends output messages to the user and also facilitates interactions among the available services.

The following is a typical sequence of events that can take place when making a reservation using the above services. The customer makes a travel reservation by sending a request through the On-line User Interface Service to the Flight Reservation and Purchase System. The request is received by the Flight Reservation and Purchase System which may specify the type of airline (for example, KLM, Delta or Air Canada), the location and the time of travel as well as the details of the hotel the customer wants. The Flight Reservation and Purchase System checks for
the availability of a flight based on the information provided by the customer and
a given system requirement, and returns an offer to the customer if available; oth-
erwise a failure message is generated. If an offer is sent to the customer, he or she
may decide to accept or cancel the offer. In the event that the customer accepts to
purchase the ticket, then the Flight Reservation and Purchase System proceeds to
check the customer’s credit card or authenticates his or her debit card or any other
means of payment and finally transfers an appropriate amount of funds from the
client’s bank account to the airline’s bank account. Composing these services must
take into account certain business constraints such as the following: (1) the Hotel
service should not be booked if the flight is not available, (2) the client can make
payment using either a debit card or a credit card but not money order payment, (3)
the composed service should process only flight reservations involving KLM or Delta
Airlines but not Air Canada, and (4) the customer must accept the offer before his
or her bank account is charged.

3.2 Service and Supervisory Control Theory Representation

In this section, we first give a quick description of the two industry standard languages
used in specifying services, namely WS-BPEL and WSDL. Next, we present the
formal details of a variant of a Labelled Transition System equipped with guards and
data variables (hereafter referred to as a Service Labelled Transition System (SLTS)).
The specification of an SLTS allows us to model and manipulate data conveniently,
and to support compact representation. That is, it is sufficient to specify systems or
processes that store and exchange data information.
3.2. SERVICE AND SUPERVISORY CONTROL THEORY
REPRESENTATION

3.2.1 Specification of Services using the WS-BPEL and the WSDL Languages

In Section 2.1.4, we introduced the WS-BPEL [5,6,35,44,150] language as the de-facto industrial language for specifying the behaviour of Web services. In this dissertation, we assume that the Web services to be composed are given as WS-BPEL processes. In the WS-BPEL language, a process can be specified using a set of basic activities (e.g., receive, reply, invoke and assign) and structured activities (e.g., while, if, pick, sequence and flow). The WSDL language (introduced in Section 2.1.4) is used to specify the operations, portTypes, messages and types provided by a Web service. Table 3.1 and Table 3.2 summarize the most relevant constructs of WSDL and WS-BPEL, respectively.

3.2.2 Formal Specification of Services: The Service Labelled Transition System (SLTS)

In this dissertation, a Web service is formally represented as a Service Labelled Transition System. In the literature this formal language is often referred to as a guarded automaton [67]. It is essentially an Extended Finite State Machine (EFSM) [157] without an update function or an action language. The formal model we present here consists of a set of states which model the dynamism of a system. The evolution of the system from one state to another is determined by its current state and the evaluation of a guard of a transition. We consider a set of data variables or data parameters \( V \) over a given finite domain \( D \). Guards are predicates or Boolean expressions over data variables. We denote the set of predicates or Boolean expressions by
### Summary: WSDL Constructs

<table>
<thead>
<tr>
<th>Element</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
<td>The WSDL types element serves as a container that envelops data type definitions. The XML schema language (XSD) is used as the type system for specifying these types.</td>
</tr>
<tr>
<td>message</td>
<td>The WSDL message element defines an abstract typed definition needed to perform an operation. A message may contain one or more logical attributes called part. Each part attribute is associated with either an element or a types attribute which are defined in the type element of the WSDL file.</td>
</tr>
<tr>
<td>portType</td>
<td>The portType specifies an abstract set of operations. It also describes the input and output messages.</td>
</tr>
<tr>
<td>operation</td>
<td>This element provides an abstract specification of the SOAP actions which the service provides. It also describes how the messages used in the service are specified.</td>
</tr>
<tr>
<td>Binding</td>
<td>This element specifies the interface protocol and how a specific portType is represented. It also defines the operations.</td>
</tr>
<tr>
<td>port</td>
<td>The port element specifies how the resources provided by a Web service can be accessed by specifying a specific endpoint using a series of bindings and the relevant network information.</td>
</tr>
<tr>
<td>service</td>
<td>A service specifies a set of related endpoints (and an endpoint is defined using a binding and an address).</td>
</tr>
</tbody>
</table>

Table 3.1: WSDL Constructs [1,35]

\( \mathcal{B} \), which is evaluated with respect to the valuation function \( f_d : \mathcal{B} \rightarrow \{true, false\} \).

We distinguish among three kinds of events, namely, \textit{input actions}, which represent the reception of messages, \textit{output actions}, which represent messages sent to external services, and \textit{atomic actions} which may modify the value of a variable arbitrarily. More specifically, the events or actions of an SLTS consist of the following:

- **Input and Output Messages**: We denote a reception of a message as \( ?m(\overrightarrow{x}) \)
### Summary: WS-BPEL Constructs

<table>
<thead>
<tr>
<th>Activity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>partnerLink</strong></td>
<td>The external Web services with which a WS-BPEL process interacts are modeled as a <em>partnerLink</em>. Each <em>partnerLink</em> is characterized by a <em>partnerLinkType</em> and a specification of the partners involved which is called <em>role</em>. The <em>partnerLinkType</em> construct declares how two parties can interact and what functionality each party provides. The <em>partnerLinkType</em> is defined in the WSDL of the external Web Service. The role of a <em>partnerLink</em> can be specified as <em>myRole</em> or <em>PartnerRole</em>. The WS-BPEL process role itself is specified by a <em>myRole</em> attribute and the <em>partnerRole</em> attribute is used to indicate the role of the partner.</td>
</tr>
<tr>
<td><strong>variable</strong></td>
<td>To model and store data, the WS-BPEL <em>variable</em> construct is used to hold data at a given state of a process. This construct allows the process to store messages that are exchanged between the business process and its partners. Variables also allow the process to store data internally that is only state based and never send to partners. The WS-BPEL specification provides three types of attributes that can be used to specify the type of a variable. These are WSDL message type, XML Schema simple type and XML Schema element.</td>
</tr>
<tr>
<td><strong>reply</strong></td>
<td>A <em>reply</em> activity is used to generate a response to a request or to a message that was received through a <em>receive</em> activity. This construct is useful when specifying a WS-BPEL synchronous process.</td>
</tr>
<tr>
<td><strong>receive</strong></td>
<td>A <em>receive</em> activity corresponds to a reception which specifies the partner link from which to receive information. It also specifies the port type and operation through which a partner can invoke it. To store the content of the incoming message a <em>variable</em> attribute is defined within the declaration of the <em>receive</em> activity.</td>
</tr>
<tr>
<td><strong>invoke</strong></td>
<td>The <em>invoke</em> operation is used to invoke an operation in the service interface. The <em>invoke</em> operation can be specified as a one-way (asynchronous) or as request-response (synchronous) using a port construct supported by the service. This construct supports two types of variables: these types are the input and output variables. In a request-response invocation both the input and output variables are specified, while in the one-way invocation, it is only necessary to specify the input variable.</td>
</tr>
<tr>
<td><strong>assign</strong></td>
<td>This constructs allows a WS-BPEL process to manipulate data. The data manipulation can include copying the contents of one attribute to another. The copy operation provides a method to copy data between variables, expressions expressed in XPath language, partner link endpoints references, and other elements.</td>
</tr>
<tr>
<td><strong>sequence</strong></td>
<td>A <em>sequence</em> activity is used to define a collection of activities to be performed sequentially. The execution of activities contained in the <em>sequence</em> activity is based on the way they are arranged with the declaration of the <em>sequence</em> activity.</td>
</tr>
<tr>
<td><strong>if</strong></td>
<td>This construct is used to specify conditional behaviour which performs different actions based on different conditional branches. The <em>if</em> branch is taken if a specified condition is <em>true</em>. The <em>else</em> branch is taken if the same condition is <em>false</em>. The <em>if</em> activity also provides an optional <em>if-else</em> branch that can be used to specify a new condition to evaluate, if the first <em>if</em> condition turns out to be <em>false</em>.</td>
</tr>
<tr>
<td><strong>while</strong></td>
<td>The <em>while</em> activity is used to specify the repeated execution of the given set of activities within the <em>while</em> loop as long as a specified condition is true. The <em>while</em> activity terminates once the loop condition no longer holds true.</td>
</tr>
<tr>
<td><strong>pick</strong></td>
<td>The <em>pick</em> activity models which activity is to be executed following the occurrence of a given event associated with this activity. An event that could trigger the execution of a <em>pick</em> activity could be an <em>onMessage</em> event or an <em>onAlarm</em> event. An <em>onMessage</em> event could be the arrival of some message in the form of the invocation of an operation while an <em>onAlarm</em> event is a timer-based alarm. The branch whose condition (i.e., <em>onMessage</em> or <em>onAlarm</em>) is satisfied first is executed.</td>
</tr>
</tbody>
</table>

Table 3.2: WS-BPEL Constructs [5, 6]
and an emission of a message as $\text{!}m(\overrightarrow{x})$, where $m$ is the name of the message also known as the 
message header, and $\overrightarrow{x}$ is the set of data parameters or variables. The symbols $?$ and $!$ are used to denote the direction of messages, respectively. Variables are local to a service and only one service can modify a variable.

- **Atomic Operations**: Operations such as function invocations are denoted by $\text{nameOperation}(I :: O)$ with input parameters $I$ and output parameters $O$. Atomic operations are indivisible functions that can modify the variables in a service. The atomic operations/functions that we consider here are similar to atomic processes defined in OWL-S and BPEL, which can access and modify the variables of a Web service. OWL-S defines an atomic process as a non-decomposable Web-accessible program. It is executed by a single (e.g., http) call, and returns a response. It does not require an extended conversation between the calling program or agent. We assume that these atomic functions are local to a given service. The atomic operation, e.g., $\text{function}(i_1, i_2 :: o_1, o_2)$ takes as inputs a set of variables $\{i_1, i_2\}$ and returns a set of variables $\{o_1, o_2\}$ as output. The effects that the atomic operation has on its output variables are not visible to the entire system. It can be observed that there are many cases where it will not make sense to assume static information about Web services. In a dynamic environment, Web services information may change while the Web service procedure is operating at runtime. Typical examples are the following: whether a product is in stock, how much it will cost or how much has been bid for it, what the weather is like, what time a train or
airplane will arrive, what seats are available for an airplane or a concert, what
shipping facilities are available for a shipping request, all of which are unknown
before runtime. The output of an atomic function in our model depends on the
time and the circumstances of invocation. That is, the output of the atomic
operation depends not only on the available input, but also on the current
state of the whole system. In addition, we assume that the service providers
keep details about the atomic operations secret. For example, if a service
solves sophisticated routing problems, the service provider does not want the
description of the service to reveal how the results are computed. Due to the
runtime-dependent nature of atomic operations, we treat them as black-box
events.

Formally, we define an SLTS as follows:

**Definition 3.2.1.** A Service Labelled Transition System is modeled by a tuple
\[ G_W = (S, S^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, S^F, V, B) \] where

- \( S \) is a finite set of states
- \( S^0 \subseteq S \) is the set of initial states
- \( \Sigma = \mathcal{I} \cup \mathcal{O} \cup \mathcal{A} \) is the set of events (i.e, set of actions), where \( \mathcal{I}, \mathcal{O}, \mathcal{A} \) denote the set of input messages (\(?m(\overrightarrow{x})\)), outputs messages (\(!m(\overrightarrow{x})\)) and atomic operations, respectively, and \( \mathcal{I} \cap \mathcal{O} = \emptyset, \mathcal{I} \cap \mathcal{A} = \emptyset, \mathcal{O} \cap \mathcal{A} = \emptyset \)
- \( \Gamma \subseteq S \times (\mathcal{I} \cup \mathcal{O} \cup \mathcal{A}) \times B \times S \) is the transition relation
- \( S^F \subseteq S \) is the set of final states
• $V = \{v_1, ..., v_n\}$ is a finite set of data variables over a given domain $D = D_1 \times D_2 \times \ldots \times D_n$.

• $B$ is the set of predicates called guards over a subset of the variables in $V$.

We employ infix notation and we write $s \xrightarrow{e|g} s'$ as shorthand for $(s, e, g, s') \in \Gamma$. A tuple $\lambda \in \Gamma$, $\lambda=s \xrightarrow{e|g} s'$ is a transition in $G_W$, where $e \in \Sigma$, and $g$ is a guard in $B$, a condition or a predicate defined over variables and formulas. The absence of an explicit guard on a transition means that the condition is always true. The dynamics of an SLTS depends on the current state of the system and on the valuation of the transition guards with respect to the current value of a variable. In the sequel, let $\Sigma^*$ denote the set of all finite strings of the form $\alpha_1\alpha_2\ldots\alpha_n$ of events from $\Sigma$, including the empty string $\epsilon$.

The possible behaviour of an SLTS is modeled by the set of executions. The execution or the run of an SLTS is a sequence of transitions $r = s_0 \xrightarrow{\alpha_0|g_0} s_1 \xrightarrow{\alpha_1|g_1} \ldots \xrightarrow{\alpha_{n-1}|g_{n-1}} s_n$ such that $\forall i < n$, $s_i \xrightarrow{\alpha_i|g_i} s_{i+1} \in \Gamma$ and the trace of the run is given by $\alpha_0\alpha_1\alpha_2\ldots\alpha_{n-1}$. An SLTS may contain both finite and infinite runs. The language generated by an SLTS, denoted by $L(G_w)$, is the set of words $L(G_w) = \{ w \in \Sigma^*| s_0 \xrightarrow{w|b} y, \ s_0 \in S^0, \ y \in S \}$ where $s_0 \xrightarrow{w|b} y$ denotes a multi-step transition relation which is defined inductively as a finite sequence of applications of a transition relation which produces a state $y$ that to which a sequence of events leads from the initial state $s_0$ and where $b$ denotes a sequence of guards.

The formal language that we define here differs from extended finite state automata [157] in that we do not require an update function. Hence, in some cases it becomes impossible to track the values of variables in our model. That is, there is no
3.2. SERVICE AND SUPERVISORY CONTROL THEORY
REPRESENTATION

action language, but we assume the updating of variables is done internally, which makes it difficult to track the values of variables.

We distinguish between two kinds of transitions, the first one is a static transition \( s \xrightarrow{e} s' \) which does not depend on a variable; the guard of this transition is always true and is triggered when the event on the transition takes place. The second type of transition is a dynamic transition \( s \xrightarrow{e[g]} s' \) which has non-trivial guards containing variables. This kind of transition is fired only if the guard on the transition evaluates to true and the event has occurred. The way the terminology static has been used in this thesis differs from how it is usually used in other areas such as static analysis. For example, in Figure 1.1(a), the transition \( s_1 \xrightarrow{\text{!}f\text{Request}\((\text{date},\text{loc})\)} s_2 \) is a static transition whereas the transition \( s_4 \xrightarrow{\text{processBooking}()[\text{av=KLM} \lor \text{av=AirCanada} \lor \text{av=Delta}]} s_5 \) is a dynamic transition. We can define an ordering on the guards of an SLTS \(^1\) to as follows.

**Definition 3.2.2. Subguards**

Let \( g_1 \) and \( g_2 \) be two guards. We call \( g_2 \) a subguard of \( g_1 \) denoted by \( g_2 \leq g_1 \), if \( g_2 \) is stronger than \( g_1 \), i.e., \( g_1 \land g_2 \iff g_2 \).

In order to model behaviours common to two or more SLTSs, we define an operation product in such a way that an event can be executed only if it is contained in all the SLTSs involved. This definition we provide will allow us to model multiple requirements of a system. For example, given \( LTS_1 \) and \( LTS_2 \) specifying certain given system requirements, an event is allowed to occur only if it is allowed in both SLTSs.

\(^1\)see similar work on extended finite state machine [149]
3.2. SERVICE AND SUPERVISORY CONTROL THEORY
REPRESENTATION

Definition 3.2.3. **Product**

Given two SLTSs $G_{W_1} = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S_1^{\mathcal{F}}, V_1, B_1)$ and $G_{W_2} = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S_2^{\mathcal{F}}, V_2, B_2)$ their product is given by $G_{W_1} \times G_{W_2} = (S_1 \times S_2, S_0^1 \times S_0^2, I_1 \cup I_2, O_1 \cup O_2, A_1 \cup A_2, \Gamma_1 \times \Gamma_2, S_1^{\mathcal{F}} \times S_2^{\mathcal{F}}, V_1 \cup V_2, B_1 \cup B_2)$ such that the transition relation $\Gamma_1 \times \Gamma_2$ is defined as follows.

- $(s_1, s_2) \xrightarrow{\alpha[g]} (s'_1, s'_2) \in \Gamma_1 \times \Gamma_2, \alpha \in \Sigma_1 \cap \Sigma_2$ if $s_1 \xrightarrow{\alpha[g]} s'_1 \in \Gamma_1$ and $s_2 \xrightarrow{\alpha[g]} s'_2 \in \Gamma_2$,
- Undefined otherwise

That is, the product of two SLTSs captures the intersection of their behaviours.

Analogous to the standard SCT where the legal language is a sublanguage of the plant, in our framework we use the notion of a simulation relation to describe the relationship between a system (plant) and a given specification, both modeled as SLTSs.

Definition 3.2.4. **Simulation Relation with Guards**

Given two SLTSs $G_{W_1} = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S_1^{\mathcal{F}}, V_1, B_1)$ and $G_{W_2} = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S_2^{\mathcal{F}}, V_2, B_2)$, $G_{W_2}$ simulates $G_{W_1}$ if there exists a relation $R \subseteq S_1 \times S_2$ such that $\forall (s_1, s_2) \in R$, if $s_1 \xrightarrow{\alpha[g]} s'_1 \in \Gamma_1$ then there exist $s'_2$ and $g'$ such that $s_2 \xrightarrow{\alpha[g']} s'_2 \in \Gamma_1$ and $g_1 \leq g'$ and $(s'_1, s'_2) \in R$.

That is, we say $G_{W_2}$ simulates $G_{W_1}$ denoted by $G_{W_1} \preceq G_{W_2}$ if every transition taken by $G_{W_1}$ can be matched by a corresponding transition in $G_{W_2}$. In essence, when the two SLTSs are represented as their respective execution trees, $G_{W_1} \preceq G_{W_2}$ means that $G_{W_1}$ is a subtree of $G_{W_2}$. 
3.2. SERVICE AND SUPERVISORY CONTROL THEORY
REPRESENTATION

3.2.3 WS-BPEL as an SLTS

In our framework, given a Web service described in the WS-BPEL language, we have implemented a translator which automatically extracts the SLTS model from the Web service specification. Formally, we model a service as an SLTS as defined above. We also assume that the available Web services are published and reside in a repository in which we select the required services that meet a given functionality. The focus of this document is on the problem of automated Web service composition, thus the issue of discovery and the selection of services are beyond the scope of this thesis.

In more detail, given a WS-BPEL specification of a process, a translation technique (see Chapter 4) is used to systematically translate this process into a corresponding SLTS. For instance, the WS-BPEL basic constructs such as receive, reply, invoke (or assign) are translated into the input, output and atomic operation transitions of a corresponding SLTS, respectively, while the structured constructs (while, if, flow, pick) are mapped into the conditions or guards of their respective transitions of the SLTS in a systematic fashion.

We model input messages and atomic functions as uncontrollable since we cannot control inputs from the user. We assume that no service can deny an input action from other services, while it is completely up to the service to control its outputs. Atomic functions are considered uncontrollable because their internal computations cannot be controlled. On the other hand, output messages from the system are modeled as controllable. In the context of Web services a guard (g) represents the preconditions on variables.
Figure 3.1 shows the SLTS representations of the four component services of the Flight Reservation and Purchase System example introduced in Section 3.1.

3.3 The SASCA Framework

3.3.1 Controller Synthesis for Service Composition

In this section, based on the representation of services using SLTSs, we formalize the problem of composing Web services, and we describe its solution by means of supervisory control theory of discrete-event systems. Our model of synthesized Web services relies deeply on message passing, interaction with data and actions. The composition problem that we consider here is as follows: given a set of available services $G_{W_1}, G_{W_2}, \ldots, G_{W_n}$ and a set of specifications $\mathcal{T}^W$ representing the goal (or desired) service over the same environment (same set of atomic actions), we would like to construct an SLTS $\mathcal{C}$, called a controller satisfying some nonblocking (see Definition 3.3.9 below) constraints which interacts with the available services to satisfy the specification $\mathcal{T}^W$. Thus, $\mathcal{C}$ serves as a controller that restricts the system in such a way that all its executions satisfy $\mathcal{T}^W$ and so that $\mathcal{C}$ is minimally restrictive (see Definition 3.3.10 below). In addition to requiring that the generated controller satisfies the controllability and nonblocking criteria, the controlled system is also free of errors that may result from communication among component services. We assume that both the available services and the goal service are expressed as SLTSs as previously defined.

In the basic design of the SASCA framework, the set of component Web services specified in WS-BPEL are supplied as inputs to the system and the requirements
3.3. THE SASCA FRAMEWORK

Figure 3.1: Available Component Services for Flight Reservation and Purchase System
are specified as SLTSs. A translator is used to generate the SLTSs representations from the WS-BPEL descriptions of the available services. Various important internal representations and manipulations of these inputs are applied from when the input enters the system to when a controller is generated. One of these internal manipulations in the framework is an intermediate preprocessing step of the plant to achieve a more refined model suitable for composition synthesis. The final output of the synthesis is a WS-BPEL executable file. In the rest of this section, we discuss the core details of our approach including relevant definitions and theorems.
3.3.2 Asynchronous Communication and System to be Controlled

In our formalism, we use asynchronous communication to model the interaction among the available services. Synchronous semantics require that during a message exchange, the sender and the receiver have to synchronize the send and receive actions, and the sender blocks until a reply is received. However, in the domain of Web services where component services are dynamically discovered and plugged in to obtain a composite service (loosely coupled), using synchronous semantics may go a long way to limit the applicability of our model. Hence, in this thesis we assume that Web services interact in an asynchronous fashion. Asynchrony can be achieved by employing unbounded memory to store the variables and parameters exchanged among component services. However, in this work the way we model service interactions does not take into consideration how the messages are stored and retrieved. Asynchrony eliminates the situation where the sender halts its process and waits for a reply from the receiver. The asynchronous semantics that we adopt here make implementation easier compared to synchronous semantics, however, it is very hard to reason about communication systems modeled using asynchronous semantics. In general, modeling the composition of communicating systems could result in various undesirable behaviours such as unspecified receptions and non-executable interactions of the system [28,178]. We will refer to these undesirable properties (unspecified receptions and non-executable interactions) as communication design errors.

The framework we propose has two inputs as shown in Figure 4.1, the composition requirements $\mathcal{T}^W$ and the set of component Web services with SLTSs as $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$. The set of available services $\mathcal{G}_{W_1}, \mathcal{G}_{W_2}, \ldots, \mathcal{G}_{W_n}$ evolves independently, but
3.3. THE SASCA FRAMEWORK

together they form a combined system $G_W$ whose behaviours we need to control. The individual component services cannot communicate among themselves; in order to exchange messages a controller is generated to mediate the interactions among component services. In the supervisory control domain, $G_W$ models the plant which represents the set of possible behaviours. As a first step in the composition process we obtain $G_W$ by combining the set of available services whose SLTSs is given by $G_{W_1}$, $G_{W_2}$, ..., $G_{W_n}$ by means of an asynchronous parallel composition which captures the notion of asynchronous communication [76, 77, 111]. The definition of asynchronous parallel composition presented below is similar to the shuffle operation in DES.

**Definition 3.3.1. Asynchronous Parallel Composition**

Given two SLTSs $G_{W_1} = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S_F^1, V_1, B_1)$ and $G_{W_2} = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S_F^2, V_2, B_2)$ their asynchronous parallel composition is given by $G_{W_1} \parallel G_{W_2} = (S_1 \times S_2, S_0^1 \times S_0^2, I_1 \cup I_2, O_1 \cup O_2, A_1 \cup A_2, \Gamma_1 \parallel \Gamma_2, S_F^1 \times S_F^2, V_1 \cup V_2, B_1 \cup B_2)$ such that the transition relation $\Gamma_1 \parallel \Gamma_2$ is defined as follows.

- $(s_1, s_2) \xrightarrow{\alpha_{[g_1]}} (s'_1, s_2) \in \Gamma_1 \parallel \Gamma_2$, if $s_1 \xrightarrow{\alpha_{[g_1]}} s'_1 \in \Gamma_1$
- $(s_1, s_2) \xrightarrow{\alpha_{[g_2]}} (s_1, s'_2) \in \Gamma_1 \parallel \Gamma_2$, if $s_2 \xrightarrow{\alpha_{[g_2]}} s'_2 \in \Gamma_2$
- Undefined otherwise

Definition 3.3.1 can be extended to $n$ services by observing that it is associative, i.e., $(G_{W_1} \parallel G_{W_2}) \parallel G_{W_3} = G_{W_1} \parallel (G_{W_2} \parallel G_{W_3})$. Therefore, without ambiguity we can write $G_{W_1} \parallel G_{W_2} \ldots \parallel G_{W_n}$ to represent the composition of multiple Web services. The definition of asynchronous parallel composition given above allows individual services to make independent moves. Definition 3.3.1 describes all possible behaviours
3.3. THE SASCA FRAMEWORK

of a given set of available services. We assume that the available services do not interact among themselves; any form of communication is through the supervisor. Hence, we require that the input (output) messages of a service are disjoint from the inputs (output) of another service. Atomic operations are local to a service. Generally, variables of a service have local scope and hence, each service refers to different internal variables. In cases where the names of variables conflict among the components services, we assume appropriate relabeling of variables will be made to resolve the conflicts. Figure 3.2 illustrates Definition 3.3.1 with an example without guards on transitions. The SLTS in Figure 3.2(c) represents the asynchronous composition of SLTSs in Figure 3.2(a) and Figure 3.2(b).

Given a set of available services, forming the asynchronous parallel composition \( G_W = G_{W_1} \parallel G_{W_2} \parallel G_{W_n} \) could result in communication errors. This definition leads us to the second step of our composition process in the next section.

3.3.3 Preprocessing Design Errors

Applying Definition 3.3.1 to combine the available services may result in two main communication design errors: cases where messages are sent to a service but it is unable to receive them and cases where a service expects a message which another service is unable to provide. That is, given the system to be controlled which represents the asynchronous parallel composition of available services \( G_W = G_{W_1} \parallel G_{W_2} \parallel \cdots \parallel G_{W_n} \), \( G_W \) may contain the following errors as defined below. We want the combined set of services to be free from communication design errors. Consider
for instance an airline service that provides several functionalities and is able to receive requests to book different kinds of flights from customers including book_KML, book_Air_Canada, book_Ethiopian_airline and so on. In this situation, the airline service may be providing more functionalities than what a particular client service actually needs. Therefore, it will be necessary to make the airline service cooperate with the client service by restricting its set of messages to a subset of the client’s request. In the following, we adopt the definitions of unspecified reception and non-executable interactions in the text by Zafiropulo et al. [178], but we tailor these definitions which were defined on FSMs to SLTSs.
3.3. **THE SASCA FRAMEWORK**

An unspecified reception is a situation where one service can send a message at a reachable state, but other services are not able to receive it. That is, the SLTS description of a service contains an emission that cannot be consumed by the related component services involved in the composition. Consider Figure 3.3(a) and Figure 3.3(b), $S_1$ and $S_2$ can communicate based on the message headers $requestFlight$ and $flightOffer$, however when $S_2$ is in state $s_1$, it is capable of sending an additional message $!searchFlight$ which cannot be consumed by $S_1$. In the following, we formally define unspecified receptions.

**Definition 3.3.2. Unspecified receptions**

Given an SLTS $G_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ of a Web service, a transition $s_i \xrightarrow{m(x)[g_i]} s_{i+1} \in \Gamma$ is an unspecified reception, if for a given execution or run $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \ldots s_i \xrightarrow{m(x)[g_i]} s_{i+1} \ldots s_{n-1} \xrightarrow{\alpha_{n-1}[g_{n-1}]} s_n$ of $G_W$ there is no $j > i$ such that $s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \in \Gamma$.

Similarly, non-executable interactions refer to a situation in which one service is able to receive a message that has not already been sent by some other service. This results in additional unmatched receptions. For instance, in Figure 3.3, when $S_1$ and $S_3$ are combined by asynchronous parallel composition, the combined system will be stuck at a state in which $S_1$ will be waiting on the reception of $?flightOffer$ at state $s_1$ while $S_3$ will be waiting on either the reception of $?searchFlight$ or $?requestHotel$ which is not being sent by any of these services.

**Definition 3.3.3. Non-executable interactions**

Given an SLTS $G_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ of a Web service, a transition
3.3. THE SASCA FRAMEWORK

$s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \in \Gamma$ is non-executable interaction if for a given execution or run $r = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \ldots s_j \xrightarrow{?m(x)[g_j]} s_{j+1} \ldots s_{n-1} \xrightarrow{\alpha_{n-1}[g_{n-1}]} s_n$ of $\mathcal{G}_W$ if there is no $i < j$ such that $s_i \xrightarrow{\alpha_i[g_i]} s_{i+1} \in \Gamma$.

That is, if there is a transition in the trace that expects an input message ($?m(\overrightarrow{x})$) but there is no corresponding output message ($!m(\overrightarrow{x})$) on a transition that precedes the input message.

**Definition 3.3.4. Communication-Error Free SLTSs**

An SLTS $\mathcal{G}_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ is said to be communication-error free if it is free from unspecified receptions and non-executable interactions.

In order to ensure freedom from non-executable interactions and freedom from unspecified receptions in the composition system, we perform a prefiltering step as part of our composition generation process to refine the system to be controlled. Given the SLTS $\mathcal{G}_W$ representing asynchronous parallel composition of the available services, we perform a refinement or preprocessing on $\mathcal{G}_W$ to get a communication-error free SLTS. We denote a plant which is a valid SLTS or the refined plant as $\text{Ref}(\mathcal{G}_W)$. This preprocessing step removes all paths that contain any unspecified and non-executable interaction errors from the original plant. This preprocessing does not affect the functionality of the system, since we want to consider a communication-error free set of communicating services. That is, once the services are composed, input (respectively, output) messages that are not consumed by other services will become useless and may obstruct system progress.

We end this section with the following assumptions on the communication medium
upon which our framework operates: (i) We do not assume the existence of any specific technique for message queuing and buffering and we assume that asynchronous communication is correctly implemented such that all problems and complications associated with asynchrony have been properly tackled; (ii) We assume that the services we model do not result in nontermination during communication. That is, the services that we consider are guaranteed to terminate during execution and as such, no service can keep sending a message infinitely and no service will wait for the reception of a particular message forever. Even though we model loops (e.g., self-loops)
in our framework, we assume that after some finite number of iteration of a loop at a given state of a service, the system will eventually progress to a final state or to a state where it terminates; and (iii) We assume a faultless communication medium, insofar as no unexpected designed errors and inconsistency such as message losses in channels and delay in communication are considered.

### 3.3.4 Composition Requirements

The composition requirements are also given as SLTSs which specify the possible accepted interactions that must hold in the composition. We require that the system requirements to be satisfied be clearly specified in terms of its input/output messages and atomic operations that would be made available to other services. We also assume that the SLTSs of the specification are also communication-error free by Definition 3.3.4. There could be multiple specifications. In that case, we use the product operation to put them together. Intuitively, a supervisor in this case will be one that guarantees that all specifications are achieved. We assume that the designer of the specification is aware of the set of services available and must write the specification in such a way that it is simulated by the combined services. In the case that the specification cannot be simulated by the plant, then further refinement must be done.

In this framework, specification can take one of the following forms: (i) A composition requirement can specify a set of constraints on the ordering of events and actions. A typical example of these constraints in our flight booking system is that the credit card of the user must be verified by the Bank service before a booking
confirmation is delivered to the customer. Another ordering requirement could be that the Flight service must confirm flight availability before the hotel is booked. In other words, the hotel should not be booked if the flight cannot be booked. A simple SLTS specification expressing this composition requirement is depicted in Figure 3.4(a). We assume that self-loops would be used at certain states to indicate that other transitions are allowed to occur at those states. (ii) Another form of composition requirement is to specify stronger guards that limit the values that can be assigned to a variable or a data parameter from a given domain. This requirement can be used to restrict the values of a variable that can be sent or received by services. In Figure 3.4(b), the SLTS specifies that the airline service from our running example can accept reservations for only KLM and Delta \((av = KLM) \lor (av = Delta)\) but not Air Canada. This specification restricts the values of the variable \(av\). Hence, a correct composition must not allow Air Canada reservations to be made. Figure 3.4(c) also specifies the kind of payment that can be made by a client to the Flight Reservation and Purchase System. The SLTS specifies that the system can only accept payment made by credit card or debit card. (iii) One can also explicitly specify a set of forbidden states that the system should not reach during execution. For example, a specification that specifies that the cost of a product \(c\) should not exceed a limit \(m\), i.e., \(c < m\) implies that \(c \geq m\) leads to a forbidden state.

### 3.3.5 Controller Synthesis

In this section, we study how to synthesize a controller that will ensure that the system’s behaviour satisfies the given requirements. We assume that the system
3.3. THE SASCA FRAMEWORK

to be controlled is given by the asynchronous parallel composition of the available services $G_W_1 \parallel G_W_2 ... \parallel G_W_n$ and the system requirements (target service) are given by $T_W$. Now, we require that the asynchronous parallel composition of the available services simulates the goal service. That is, $T_W \preceq G_W$. If it does not simulate the goal service we perform refinement on the target services.

Given the plant $G_W$ and a specification $T_W$ SLTSS, we can obtain another SLTSS $C^0$ by refining $G_W$ with respect to $T_W$ such that the behaviour of $C^0$ is identical to
3.3. THE SASCA FRAMEWORK

\( G_W \), but the executions not allowed in \( T_W \) result in forbidden states of \( C^0 \). The definition that follows defines how this refined SLTSs can be constructed. The definition specifies the product of the plant and the specification such that the set of states of the product transitions system are partitioned in to safe (good) and forbidden (bad) states. The definition is an extended version of product that is we use later on in the text to construct a controller.

Given two SLTSs \( G_{W_1} \) and \( T_W \) representing the plant and the specification, respectively, we can compute their product as well as refine the plant with respect to the specification in such a way that sequences of transitions not allowed by the specification will end in bad or forbidden states in the plant. A bad or forbidden state is a state reachable in \( G_{W_1} \) but not in \( T_W \).

**Definition 3.3.5. Composition Refinement**

The composition refinement of the plant \( G_{W_1} = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S^F_1, V_1, B_1) \) and the specification \( T_W = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S^F_2, V_2, B_2) \), denoted by \( G_{W_1} \times_{ref} T_W \), is given by \( G_{W_1} \times_{ref} T_W = (S_1 \times (S_2 \cup \{s^{Bad}\}), S_0^1 \times S_0^2, I_1 \cup I_2, O_1 \cup O_2, A_1 \cup A_2, \Gamma_1 \times \Gamma_2, S^F_1 \times S^F_2, V_1 \cup V_2, B_1 \cup B_2) \), where \( s^{Bad} \) denotes a bad or forbidden state and the transition relation \( \Gamma_1 \times \Gamma_2 \) is defined as follows.

- \((s_1, s_2) \xrightarrow{\alpha [g_1 \land g_2]} (s_1', s_2') \in \Gamma_1 \times \Gamma_2 \) and \((s_1, s_2) \xrightarrow{\alpha [g_1 \land \neg g_2]} (s_1', s^{Bad}) \in \Gamma_1 \times \Gamma_2, \) if \( s_1 \xrightarrow{\alpha [g_1]} s_1' \in \Gamma_1 \) and \( s_2 \xrightarrow{\alpha [g_2]} s_2' \in \Gamma_2 \)

- \((s_1, s_2) \xrightarrow{\alpha} (s_1', s^{Bad}) \in \Gamma_1 \times \Gamma_2, \) if \( s_1 \xrightarrow{\alpha} s_1' \in \Gamma_1 \) and \( s_2 \xrightarrow{\alpha} s_2' \notin \Gamma_2 \)

- Undefined otherwise
Let $C^0 = G_{W1} \times_{\text{ref}} T^{W}$. The first item of the definition creates two new transitions in $C^0$ (the refined SLTS) with the same events but different guards. Intuitively, the first transition, given by $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge g_2]} (s'_1, s'_2)$, replaces the guards of $C^0$ with that of the specification and the resultant state is a state allowed by both the plant and the specification. The second transition, given by $(s_1, s_2) \xrightarrow{\alpha[g_1 \wedge \neg g_2]} (s'_1, s_{\text{Bad}})$, is essentially the same as the former, however the guard of the latter transition is $g_1 \wedge \neg g_2$ which results in a new state allowed by the plant but forbidden in the specification. The second item of the definition creates a new transition in $C^0$ if an event is allowed by the plant but not legal in the specification.

Now, the set of states of $C^0$ is given by $Y = S_1 \times (S_2 \cup \{s_{\text{Bad}}\})$. A state $(s_1, s_2) \in Y$ is said to be forbidden if $s_2 = s_{\text{Bad}}$. That is, it is a bad state. We will denote $S^\text{Bad}_{C^0}$ as the set of bad states of $C^0$. That is the set of states reachable in $G_{W1}$ but not in $T^{W}$. The state $(s_1, s_2) \in Y$ is safe if $s_2 \neq s_{\text{Bad}}$. States that are not in $S^\text{Bad}$ are called safe or good states denoted by $S^\text{Good}_{C^0}$. Now, by strengthening the guards of $C^0$ with respect to the plant so that forbidden states in $C^0$ are not reachable we obtain a new SLTS which we will call a safe SLTS of $C^0$. We will show how to strengthen the guards of $C^0$ later on in Algorithm 3.5 of Section 3.4.

We assume that the set of events $\Sigma$ is partitioned into three disjoint subsets namely, controllable events $\Sigma_c \subseteq \Sigma$, uncontrollable $\Sigma_{uc} \subseteq \Sigma$ and enforceable events $\Sigma_f \subseteq \Sigma$. Controllable events can be disabled by the controller while uncontrollable events cannot be prevented from occurring. In addition, the enforceable events are special events that can be enforced by the controller. They are able to preempt both controllable and uncontrollable events at runtime but not static transitions. The
3.3. THE SASCA FRAMEWORK

notion or the intent of control in this framework involves the following techniques. That is, the controller exerts control as follows. Firstly, the generated controller prevents the system from firing or taking a particular path that violates the control requirement and secondly, it also prevents the system from reaching states designated as forbidden. In order to achieve the above control goals the supervisor enacts control based on the following three control criteria:

1. Disabling of controllable events on a transition (static transition)

2. Assignment of stronger guards to controllable transitions (transitions whose events are controllable)

3. Enforcement of enforceable events

To develop our control synthesis algorithms and strategies, we assume that the system evolves from one state to another based on the kind of transitions (static or dynamic transitions) at a given state. Thus, it is imperative to study the kind of transitions at a given state. We will explore the notion of control based on whether the transition is static or dynamic, or whether the values of the variable used on the transition can be tracked or not. Once we have generated $C^0$ from Definition 3.3.5, we will iteratively pare down $C^0$ until it satisfies the requirements.

**Static Transition Case:** Given a static transition (i.e., a transition with the trivial guard “true”), if this transition is associated with a controllable event which is allowed by the plant $G_w$ but that violates system requirements, then we assume that this transition will be disabled by the supervisor. However, if the event associated with this transition is an uncontrollable event, then we must ensure that this static
transition does not occur in the plant. If the specification does not allow a static transition, we will not allow the system to reach a state where it can occur.

**Dynamic Transition Case (Dynamic Type 1 Transition):** Let $G_{W_1} = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S^f_1, V_1, B_1)$ and $G_{W_2} = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S^f_2, V_2, B_2)$ be two Web services, and suppose that the transition $t_1 = s_1 \xrightarrow{!m(v_1)[g_1]} s'_1 \in \Gamma_1$ is an emission of a variable $v_1$ from $G_{W_1}$ to $G_{W_2}$ and $t_2 = s_2 \xrightarrow{?m(v_1)[g_2]} s'_2 \in \Gamma_2$ is reception of $v_1$ by $G_{W_2}$. Now, if the variable $v_1$ (the content of the message $!m(v_1)$) has not been modified from its original value at the state where it was defined or last assigned until the state at which it is actually used in, then the value of $v_1$ has not changed. This implies that we can easily track the value of the variable $v_1$ in the message from the service that sent it ($G_{W_1}$) to the receiving service ($G_{W_2}$). Now, at the state that this variable is being used, if there is a condition on a transition ($t_2$) in $G_{W_2}$ from this state that imposes a restriction on the set of values the variable can take, then we need to make sure that the guard on $t_1$ is never true for those values to prevent the system from reaching an forbidden state. Hence, the supervisor can enact control by restricting the value of the variable at the sending service side before it would be received by the receiving service. The control strategy employed to deal with this kind of transition is to assign stronger guards to a controllable transition. Hence, we generate the guard $g_1 \land \neg g_2$ and attach it to the transition $t_1$. In our example above, the Bank service Figure 3.1(c) can accept a debit card, credit card or money order as a means of payment, but the specification in Figure 3.4(c) prevents a payment with money order. To satisfy this constraint, we put a condition (or strengthen the guard) on the transition of the service that will send a request for payment to the
Bank service to prevent it from sending a request for money order payment.

**Dynamic Runtime-Dependent Transition Case (Dynamic Type 2 Transition):** This case deals with atomic operations whose output value is unknown until runtime. This issue brings the concept of nondeterminism into our model. Since the values of the variable are unknown until runtime we cannot treat this case in the same way as the previous case. During design time we will classify certain events (e.g., failure message events) as enforceable events. If such a transition does not exist we will introduce a new transition into the plant and the specification. To be able to prevent transitions that may cause a specification violation and that have guards containing outputs of atomic operations, we will rely on enforceable events to preempt uncontrollable events from happening when the output of the variable from atomic operations violates a specification. That is, if a failure could occur in the system due to values associated to atomic operations, then enforceable transitions are used to preempt the failure. This construction can be done by modification of the plant [50,161]. Consider Figure 3.1(a), when the transition from state $s_2$ to $s_3$ labeled with $\text{checkAirlinesAvail}(date, loc :: av)$ has an output variable $av$ which can take KLM, AirCanada or Delta as its values. The operation $\text{checkAirlinesAvail}(date, loc :: av)$ is assumed to be black-box, so we do not know which value it will assign to $av$. Now, the specification in Figure 3.4(b) limits the values that $av$ can take to only KLM and Delta. To ensure that the transition from state $s_3$ to $s_4$ in Figure 3.1(a) is never taken when the value of $av$ is AirCanada, we mark the transition $s_3 \xrightarrow{\text{notAvail}(\cdot)} s_1$ in Figure 3.1(a) as an enforceable transition.
3.3. THE SASCA FRAMEWORK

The value of $av$ is monitored so that this enforceable transition can be used to pre-empt other transitions at runtime when the value of $av$ violates a specification. This kind of situation is very common in SOA applications. For instance, the WS-BPEL language provides certain constructs such as Fault Handler and Event Handler to deal with unexpected failures that may occur at runtime. Example 3.3.1 below further illustrates the notion of event enforcement.

Example 3.3.1. Event Enforcement

Consider the plant $G_W$ in Figure 3.5 which at the initial state can emit a message $!m_1(x)$. The emission of this message transmits a variable $x$ which is consumed by the transition from state $s_1$ to $s_2$. Now, the transition from state $s_2$ to $s_3$ uses the variable $x$ as an input to the atomic operation $atom_{op1}(x :: y)$. This atomic operation performs some internal computation and outputs the variable $y$ to be used in the next state. As stated above, all output transitions are controllable while all input and atomic operation transitions are uncontrollable.

Now, consider a specification on $G_W$, which states that every transition in $G_W$ is allowed except that the value of $y$ is constrained to the set $y \in \{\beta, \gamma\}$. Given this requirement, it means that $G_W$ must not be allowed to take the uncontrollable transition from $s_3$ to $s_5$ (i.e., $y$ cannot be equal to $\lambda$). At this point, the problem is how to get to state $s_3$ without firing the uncontrollable transition from $s_3$ to $s_5$ and also noting that (i) the value of $y$ cannot be altered until $G_W$ is in state $s_2$, and (ii) $y$ is the output of the atomic operation $atom_{op1}(x :: y)$ and that the internal computation of $atom_{op1}(x :: y)$ is not known (i.e., $atom_{op1}(x :: y)$ is a black-box and how the value of $y$ is computed is not known and cannot be modified). To this end,
all we can do is to take whatever value of $y$ that is produced by $\text{atom}_{\text{op}1}(x :: y)$ at runtime.

The solution we provide is to ensure that there is an enforceable event exiting state $s_3$ which will be used to preempt the uncontrollable transition from $s_3$ to $s_5$ at runtime. In more detail, the variable $x$ and $y$ will be monitored at runtime to ensure that once $y = \lambda$ the enforceable transition from $s_3$ to $s_7$ will be triggered to preempt the transition $s_3$ to $s_5$. If there is no enforceable event going out from state $s_3$ then at design time, we add a new enforceable transition to $\mathcal{G}_W$ which is triggered when the value of $y$ turns out to be $\lambda$.

Next, in the following definition, we define the controlled system, which is given by the resultant transition system created when the transition system of the plant is coupled with the transition system of the controller.

**Definition 3.3.6. Controlled System**

Let $\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F, V, \mathcal{B})$ be the SLTS of a given plant, and let $\mathcal{C} =$
(SC, SC0, IC, OC, AC, GC, SCF, VC, BC) be an SLTS that represents the controller of GW.

The controlled system C ⊗ GW representing the behaviour of GW when constrained (controlled) by C is given by C ⊗ GW = (SC × SC0, IC ⊔ IC, OC ⊔ OC, AC ⊔ AC, GC × GC, SCF × SCF, VC ⊔ VC, BC ⊔ BC)

where:

- (s1, s2) \xrightarrow{m[g1 \land g2]} (s'1, s'2) ∈ GC × GC

\[
\begin{cases}
  s_1 \xrightarrow{\neg m[g_1]} s'_1 \in GC \text{ and } s_2 \xrightarrow{m[g_2]} s'_2 \in \Gamma, \\
  \text{or}
  s_1 \xrightarrow{m[g_1]} s'_1 \in GC \text{ and } s_2 \xrightarrow{\neg m[g_2]} s'_2 \in \Gamma,
\end{cases}
\]

- (s1, s2) \xrightarrow{\alpha[g]} (s'1, s'2) ∈ GC × GC, \alpha ∈ AC \cup AC\text{ if } s_1 \xrightarrow{\alpha[g]} s'_1 \in GC \text{ and } s_2 \xrightarrow{\alpha[g]} s'_2 \in \Gamma,

- Undefined otherwise

In other words, a transition is possible in the controlled system if (i) either the plant (or controller) wants to send (or receive) a message and the controller (or plant) can receive (send) it, or (ii) it is an atomic operation transition and this transition is also possible in the controller transition system, which implies that the guards are true and the transition can be fired.
The following definition defines another type of simulation relation, one between the controlled system and the specification in terms of their input actions, output actions and atomic operation actions. Roughly speaking, a simulation relation between the controlled system and the specification will capture the requirement that any transitions allowed in the controlled system should be allowed in the specification.

**Definition 3.3.7.** (Controlled System and Specification) Let $m$ be the message header of the output message $!m$ and the input message $?m$. Let $\mathcal{H}$ be the set of message headers of a given SLTS. Given a controlled system $C \otimes G = (S_1, S_0, I_1, O_1, A_1, \Gamma_1, S^F_1, V_1, B_1)$ and a specification $T^W = (S_2, S^0_2, I_2, O_2, A_2, \Gamma_2, S^F_2, V_2, B_2)$ a simulation relation between $C \otimes G$ and $T^W$ is a relation $R \subseteq S_1 \times S_2$ such that $orall (s_1, s_2) \in R$,

- if $s_1 \xrightarrow{m[g]} s_1'$ in $\Gamma_1$ and $m \in \mathcal{H}$ then $\exists s_2', s_2 \xrightarrow{!m[g]} s_2'$ in $\Gamma_2$ and $(s_1', s_2') \in R$ or $\exists s_2', s_2 \xrightarrow{?m[g]} s_2'$ in $\Gamma_2$ and $(s_1', s_2') \in R$, and
- if $s_1 \xrightarrow{\alpha[g]} s_1'$ in $\Gamma_1$ and $\alpha \in A_1$ then $\exists s_2', s_2 \xrightarrow{\alpha[g]} s_2'$ in $\Gamma_2$ and $(s_1', s_2') \in R$.

### 3.3.6 Controllability

The original setting of supervisory control theory considers *language-based controllability* [33, 172] which assumes the underlying automata to be deterministic. Language-based controllability was subsequently extended to provide a stronger notion of controllability called *state-controllability* [60, 113, 182] for nondeterministic discrete-events systems. In this work, we rely on a notion similar to state-based controllability to capture the concept of controllability of SLTS in our model.
3.3. THE SASCA FRAMEWORK

For a given SLTS $T^W$ specification which is simulated by a given plant $G_W$, a reachable state $(p, q)$ in $G_W \times T^W$ is uncontrollable if the following holds: (1) if an uncontrollable transition labeled with $\alpha$ (static transition) can be fired from the state $p$ in the plant but not from the state $(p, q)$ in $G_W \times T^W$; and (2) if an uncontrollable transition labeled with $\alpha$ and a guard $g_1$ can be fired in $G_W$ at state $p$ and this same uncontrollable transition labeled with $\alpha$ but a different guard $g_2$ is possible at $(p, q)$ in the $G_W \times T^W$, then whenever $g_1$ evaluates to true for a given set of values of a variable $v$, $g_2$ does not always evaluate to true for the same values of $v$. That is, $g_1$ does not imply $g_2$. This implies that the uncontrollable transition at $(p, q)$ leads to a forbidden state.

Let $s \overset{\delta}{\rightarrow}$ denote that there exists at least one state $s'$ such that $s \overset{\delta}{\rightarrow} s'$ and denote $E^S_p(s) = \{ \delta \in \Sigma \mid s \overset{\delta}{\rightarrow} \}$ as the set of enabled static transitions of the state $s \in S$ of the SLTS $P$. Similarly, let $s \overset{\delta[g]}{\rightarrow}$ denote that there exists a guard $g$ and at least one state $s'$ such that $s \overset{\delta[g]}{\rightarrow} s'$ and let $E^D_P(s) = \{ (\delta, g) \mid s \overset{\delta[g]}{\rightarrow}, \delta \in \Sigma, g \in B \}$ represent the set of dynamic transitions enabled at state $s$ of the SLTS $P$.

**Definition 3.3.8. Controllability**

Given two SLTSs $G_W = (S_1, S_0^1, I_1, O_1, A_1, \Gamma_1, S_1^F, V_1, B_1)$ and $T^W = (S_2, S_0^2, I_2, O_2, A_2, \Gamma_2, S_2^F, V_2, B_2)$, representing the plant and the specification, respectively, such that $T^W \preceq G_W$. A state $(p, q) \in S_1 \times S_2$ is controllable if the following holds:

1. Static Controllability:

$$\forall \delta \in \Sigma_{uc} : \delta \in E^S_{G_W}(p) \implies \delta \in E^S_{(G_W \times T^W)}((p, q))$$
2. **Dynamic Controllability:**

\[ \forall \delta \in \Sigma_{uc} : (\delta, g_1) \in E_{GW}^{D}(p) \implies [\exists g_2 : (\delta, g_2) \in E_{(GW \times TW)}^{D}((p, q)) \land (g_1 \implies g_2)] \]

\[ \text{or } [\exists \delta', \exists g_3 : (\delta', g_3) \in E_{(GW \times TW)}^{D}((p, q)) \land \delta' \in \Sigma_f] \]

A plant \( GW \) is said to be *state controllable with respect to TW* if all reachable states of \( GW \times TW \) are state controllable. According to Definition 3.3.8, uncontrollable transitions that are enabled in the reachable states of the plant state \( q \) by following the same trace, must also be enabled at the corresponding reachable state \((p, q)\) of \( GW \times TW \). Thus, we say \( GW \) is controllable if: (1) \( \delta \) is uncontrollable and \( \delta \) is the current static transition event enabled in \( GW \) implies that \( \delta \) is also enabled at the corresponding state of \( GW \times TW \), and (2) \( \delta \) is a dynamic uncontrollable event and a guard \( g_1 \) is possible at the current state of \( GW \) then, it implies that there exists a corresponding uncontrollable dynamic transition and a guard \( g_2 \) in \( GW \times TW \) such that \( g_2 \) is true only if \( g_1 \) is true, or there exists an enforceable transition \( \delta' \) that can preempt any uncontrollable transition in the current enabled state.

Now, let \( g_n^A(v) \) denote a guard \( g_n \) with a variable \( v \) whose values depend on an output of an atomic operation \( A \) of a given transition system (e.g., in Figure 3.6, \( A = func_1(x :: y), v = y, g_n = (y < 10) \)). We state the following corollary as a consequence of controllability of a given plant and a specification. Here, we will assume without loss of generality that a guard depends on only one variable.

Corollary 3.3.1 below states that given a plant \( GW \) and a specification \( TW \), if a
3.3. THE SASCA FRAMEWORK

![Figure 3.6](image)

dynamic type 2 transition, say $T$, is enabled at a state $p$ of the plant $G_W$ but not at a corresponding state $(p, q)$ of the specification $T^W$ then, for the state $(p, q)$ to be state controllable it implies that there must exist an enforceable transition also enabled at $(p, q)$ to preempt $T$.

**Corollary 3.3.1.** Let $G_W = (S_{G_W}, S^0_{G_W}, I_{G_W}, O_{G_W}, A_{G_W}, \Gamma_{G_W}, S^E_{G_W}, V_{G_W}, B_{G_W})$ and $T^W = (S_{T_W}, S^0_{T_W}, I_{T_W}, O_{T_W}, A_{T_W}, \Gamma_{T_W}, S^E_{T_W}, V_{T_W}, B_{T_W})$ denote a plant and its specification, respectively, and let $(p, q) \in S_{G_W} \times S_{T_W}$. If \( \exists \delta \in \Sigma_{uc}, \exists g^A(v) \in \mathcal{B} : (\delta, g^A(v)) \in E^D_{G_W}(p) \) but \( \not\exists (g^A(v))' : (\delta, (g^A(v))') \in E^D_{T_W}(p, q) \), and \( g^A(v) \land (g^A(v))' \) satisfiable, then if $(p, q)$ is state controllable then there must exist $\delta' \in \Sigma_f$ and a guard $g''$ such $(\delta', g'') \in E^D_{T_W}(p, q)$.

**Proof.** The proof follows from the second part of definition 3.3.8 where the transition is a dynamic type 2 transition with guard $g^A(v)$.

**Example 3.3.2. Illustration of Corollary 3.3.1**

Consider the plant service $G_W$ in Figure 3.7 and the target service $T^W$ of Figure 3.8. $T = s_2 \xrightarrow{\text{func}_1(x :: y) \ [y < 10]} s_3 \in \Gamma_{G_W}$ is a dynamic type 2 transition whose guard $(y < 10)$ has a variable whose values depends on the output of the atomic operation $\text{func}_1(x :: y)$.

Since $T^W$ does not allow $T$ at state $t_2$, and $G_W$ is state controllable with respect to $T^W$, then there must exist an enforceable transition $T'$ also enabled at state $s_2$ which is given by $t_2 \xrightarrow{\text{failMessg}()} t_4 \in \Gamma_{T_W}$ to preempt $T$ at runtime.
The control solution that we seek in our approach requires that the system does not reach a state from which the only exiting transitions lead to unsafe states. We formalize this in the following definition.

**Definition 3.3.9. Nonblocking**

An SLTS $\mathcal{G}_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ is nonblocking if $s \xrightarrow{g_j} s' \in \Gamma \Rightarrow \exists \delta \in \Sigma^*$ and a sequence of guards $g_j$ such that $s' \xrightarrow{\delta[g_j]} s''$ and $s'' \in S^F \setminus S^{Bad}$, where $S^{Bad}$ is the set of unsafe (bad) states.

A controller is minimally restrictive in the sense that it only disallows transitions
that must be disallowed. It is natural to require that a controller restricts the plant as little as possible. We formalize this qualitative property in the following definition using the pre-order notion implied by simulation relation.

**Definition 3.3.10. Minimally Restrictive Controller**

*Given a plant $G_W$ and a specification $T^W$, a controller $C$ for $G_W$ satisfying $T^W$ is minimally restrictive if there does not exist a controller $C'$ for $G_W$ where $C'$ satisfies $T^W$ such that $C \neq C'$ and $C \otimes G_W \preceq C' \otimes G_W$.*

The composition problem that we consider is as follows. Given a set of available services $G_{W1}, G_{W2}, \ldots, G_{Wn}$ and a set of specifications $T^W$ representing the goal service over the same environment (same set of messages and atomic actions), we would like to construct a controller $C$ such that $C \otimes (G_{W1} \parallel G_{W2} \ldots \parallel G_{Wn})$ is simulated by $T^W$ and satisfies some nonblocking constraints. Thus, $C$ serves as a controller that interacts with the uncontrolled system in such a way that all its executions satisfy $T^W$ and such that $C$ is minimally restrictive. That is, we seek to generate an SLTS which interacts with the system $G_{W1} \parallel G_{W2} \ldots \parallel G_{Wn}$ to satisfy the specification $T^W$. In addition to requiring that the generated controller satisfies nonblocking criteria, the controlled system is also free of errors that may result from communication among component services. We formalize this in the following problem statement.

**Definition 3.3.11. Composition Problem**

*Let $G_{W1}, G_{W2}, \ldots, G_{Wn}$ be a set of SLTSs and let $T^W$ be the composition requirements. The composition problem is to find a nonblocking, communication-error free and minimally restrictive controller $C$ such that $C \otimes (G_{W1} \parallel G_{W2} \ldots \parallel G_{Wn}) \preceq T^W$.***
3.4. COMPOSITION SYNTHESIS ALGORITHM

The definition implies that the controller constrains the plant such that every transition that can be taken by the controlled system $C \otimes (G_{W_1} \parallel G_{W_2} \ldots \parallel G_{W_n})$ can also be taken in the specification. The intuition is that controllability will be necessary and sufficient to solve the composition problem, as will be proven in Section 3.5.

3.4 Composition Synthesis Algorithm

In this section, we provide the core details of our approach by presenting a set of algorithms that can be used to generate a composition. The composition generation technique proposed in our framework is an incremental process. Algorithm 3.1 presents a step-by-step process that can be used to build a nonblocking and communication-error free controller for a given system that ensures that the controllability condition is met. Figure 3.9 presents a flow-chart describing the interdependence of the other algorithms called by Algorithm 3.1 as well as a legend showing which algorithm is being used at which stage. The algorithm takes the set of available component services and a goal service specifying the functional requirements as inputs.

The algorithm first refines the plant $G_W$ by removing communication design errors (i.e., non-executable interactions and unspecified receptions) which is given by Line 2. Once the plant has been transformed into its communication-error free SLTS form, we check whether the target service is simulated by the plant $T_W \leq G_W$ (Line 3). The function $\text{simulationCheck}(G_W, T_W)$ is an implementation of the simulation relation given in Definition 3.2.4. The function $\text{simulationCheck}(G_W, T_W)$ takes $G_W$ and $T_W$ as inputs, and checks if $G_W$ simulates $T_W$. It returns “true” if $G_W$ simulates $T_W$ and false if $G_W$ does not simulate $T_W$. In the case that a simulation
3.4. COMPOSITION SYNTHESIS ALGORITHM

Figure 3.9: Flow-chart Describing the Interdependence of the Algorithms
relation exists between the plant and the target service, we then make adjustments to the plant to include a special event that will enable the controller to enforce enforceable events at runtime. This step is given in Line 4 of Algorithm 3.1. The algorithm then computes the composition refinement (given by Definition 3.3.5) of the plant and the target service to get a new SLTS $C^0$ (Line 6) upon which further minimization steps will be performed.

The next step of the algorithm (i.e., the repeat until loop Lines 8-13) then performs various reductions on $C^0$. Line 10 performs static controllability minimization which is given by Algorithm 3.3. Line 11 eliminates blocking states and states from which only bad states are reachable. This line (Line 11) is implemented by Algorithm 3.4. In Line 12 of the algorithm, we compute dynamic controllability minimization and generate stronger guards to ensure that all executions of the $C^0$ lead to safe states. Lines 8-13 perform a fixed-point computation on $C^0$ and terminate when a fixed-point is reached (i.e., if $C^k == C^{k-1}$). Finally, Lines 15–18 of Algorithm 3.1 check whether $C^k$ is communication-error free or not, communication-errors may be introduced during the elimination of states and transitions. On the occasion that $C^k$ is not communication-error free, Algorithm 3.1 refines it and then moves back to Line 4. To ensure that the generated controller is able to communicate with the available services, we reverse the direction of the messages of $C^k$ in Line 19. This implies that an input message $?m(x)$ in $C^k$ will become an output message $!m(x)$ and vice versa. The message header $m$ does not change; it is only the directions of the messages that change. In the event that the algorithm does not find a simulation relation between the plant and the target service we iteratively refine
the target service until a simulation relation is found (Line 21).

Algorithm 3.1 Composition Synthesizer (Controller)

**Input:** The SLTS representing asynchronous parallel composition of the available services \(\mathcal{G}_W = (\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \Gamma, \mathcal{S}^F, \mathcal{V}, \mathcal{B})\) and a target service \(\mathcal{T}_W = (\mathcal{S}_T^W, \mathcal{S}_T^0, \mathcal{I}_T^W, \mathcal{O}_T^W, \mathcal{A}_T^W, \Gamma_T^W, \mathcal{S}_T^F, \mathcal{V}_T^W, \mathcal{B}_T^W)\)

**Output:** A communication-error free, nonblocking controller \(\mathcal{C}\) that satisfies \(\mathcal{T}_W\) for \(\mathcal{G}_W\).

1: **procedure** Composer\((\mathcal{G}_W, \mathcal{T}_W)\)
2: \(\mathcal{G}_W \leftarrow \text{removeCommunicationErrors}(\mathcal{G}_W)\) \(*\text{Algorithm 3.2}*/
3: \text{if} simulationCheck\((\mathcal{G}_W, \mathcal{T}_W)\) \text{then}
4: \(\mathcal{G}_W \leftarrow \text{plantAndSpecAdjustment}(\mathcal{G}_W)\) \(*\text{Algorithm 3.10}*/
5: \(\mathcal{T}_W \leftarrow \text{plantAndSpecAdjustment}(\mathcal{T}_W)\) \(*\text{Algorithm 3.10}*/
6: \(C^0 \leftarrow \mathcal{G}_W \times_{\text{ref}} \mathcal{T}_W\)
7: \(k \leftarrow 0\)
8: repeat
9: \(k \leftarrow k + 1\)
10: \(C^k \leftarrow \text{staticControllability}(\mathcal{G}_W, C^{k-1})\) \(*\text{Algorithm 3.3}*/
11: \(C^k \leftarrow \text{unsafeStateMinimization}(C^k)\) \(*\text{Algorithm 3.4}*/
12: \(C^k \leftarrow \text{dynamicControllabilityAndGuardGeneration}(\mathcal{G}_W, C^k)\)
13: \(*\text{Algorithm 3.5}*/
14: until \(C^k == C^{k-1}\)
15: \(C \leftarrow \text{removeUnsafeState}(C^k)\) \(*\text{Algorithm 3.7}*/
16: \text{if} \(C^k\) is not communication-error free \text{then}
17: \(C^k \leftarrow \text{removeCommunicationErrors}(C^k)\) \(*\text{Algorithm 3.2}*/
18: \text{Go to Line 7}\)
19: \text{end if}\n20: \(C \leftarrow \text{reverseMessageDirection}(C^k)\) \(*\text{Algorithm 3.8}*/
21: \text{else if} (\text{refineTargetToBeSimulatedByPlant}(\mathcal{G}_W, \mathcal{T}_W) \neq \emptyset) \text{then}\n22: \(\mathcal{T}_W \leftarrow \text{refineTargetToBeSimulatedByPlant}(\mathcal{G}_W, \mathcal{T}_W)\)
23: \(*\text{Algorithm 3.9}*/
24: \text{Go to Line 4}\n25: \text{else}\n26: \text{return null}\n27: \text{end if}\n28: \text{return } C\)
29: end procedure
Algorithm 3.2 converts a given SLTS into its communication-error free form as given in Definition 3.3.2 and Definition 3.3.3. The input to this algorithm is the asynchronous parallel composition of the available services $G_W$. Algorithm 3.2 traverses the SLTS to eliminate unspecified receptions and non-executable interactions. In Lines 22-25, given a state $s_j$ of $G_W$, if there is an output transition $t = s_j \xrightarrow{lm(x)[y_1]} s_{j+1}$ exiting state $s_j$ then the procedure findAllPaths (Lines 3-16 of Algorithm 3.2) is called upon to find all the paths or sequences of transitions from $s_{j+1}$ such that for each path a state is visited once (i.e., loops are avoided). Now, Algorithm 3.2 (Lines 26-29) checks for unspecified receptions and eliminates $t$ if none of these paths does not satisfy Definition 3.3.2 (Lines 26-31). Similarly, Lines 35-45 remove transitions that do not meet the non-executable interaction condition (Definition 3.3.3). Algorithm 3.2 Line 20-47 iterates until there are no unspecified receptions and non-executable iterations in $G_W$.

Algorithm 3.3 constructs a static controllable SLTS and iteratively creates new transitions that lead to bad states from a given SLTS. The input to this algorithm is the plant, and the composition refinement of the plant and the target service $C^k$. In the first iteration of the repeat until loop of Algorithm 3.1, $C^k$ is given by $C^0$. The set of states of $C^k$ is partitioned into safe states $S_{C^k}^{Good}$ and bad states $S_{C^k}^{Bad}$. For a given state $p$ in $G_W$ and a corresponding state in $(p, q) \in C^k$, if a static uncontrollable transition is enabled at state $p$ but not in $(p, q)$ (which is given by the first and second for loops), first, the algorithm creates a new bad state $s_{Bad} \in S_{C^k}$ and all dynamic transitions leading to $(p, q)$ are diverted to $s_{Bad}$ (Lines 8-13). This keeps the structure of dynamic transitions. Second, the uncontrollable state is eliminated
Algorithm 3.2 removeCommunicationErrors

Input: $\mathcal{G}_W = (\mathcal{S}, S^0, \mathcal{I}, \mathcal{O}, A, \Gamma, S^F, V, B)$

Output: communication-error free SLTS of $\mathcal{G}_W = (\mathcal{S}, S^0, \mathcal{I}, \mathcal{O}, A, \Gamma, S^F, V, B)$

1: Let $s$ be the starting state
2: Let $visited$ be a list of states representing the current path

3: procedure findAllPaths($\mathcal{G}_W$, $s$, $visited$)
   4: Let $PathList ← \emptyset$ be the set of all path
   5: if $s ∈ visited$ then
      6: return $PathList$
   7: end if
   8: $visited ← visited \cup \{s\}$
   9: for each successor state $s'$ such that there exist a transition $t = s \xrightarrow{\delta[g]} s'$
      10: Let $newVisited ← \emptyset$ be a list of states representing the next path
      11: $newPath ← findAllPaths(\mathcal{G}_W, s', newVisited)$
      12: $PathList ← PathList \cup newPath$
   13: end for
   14: return $PathList$
   15: end procedure

16: procedure removeCommunicationErrors($\mathcal{G}_W$)
17: \* Removal of unspecified receptions */
18: Let $\mathcal{G}_W' ← \emptyset$
19: while $\mathcal{G}_W ≠ \mathcal{G}_W'$ do
20: Let $\mathcal{G}_W' ← \mathcal{G}_W$ be an SLTS
21: for each state $s_j ∈ S$ do
22: \* continue with this procedure on next page */
23: for each outgoing output transition
24: $t = s_j \xrightarrow{m(x)[g_j]} s_{j+1} ∈ \Gamma$ exiting $s_j$
25: $pathList ← FindAllPath(\mathcal{G}_W, s_{j+1}, \Gamma)$
26: if exists no path $p = s_{j+1} \xrightarrow{a_0[g_0]} \ldots \xrightarrow{a_{j+1}[g_{j+1}]} s_{j+n+1}$ in $pathList$ such that there is a corresponding input transition $t' = s_i \xrightarrow{m(x)[g_i]} s_{i+1} ∈ p$ with $j < i$ then
27: $\Gamma ← \Gamma \backslash \{t\}$ \* removes the transition from the set of transitions of $\mathcal{G}_W$ if $t$ can not be consumed by any transition*
28: end if
29: end for
30: end for
31: end procedure
Algorithm 3.2 removeCommunicationErrors cont.

```plaintext
33: \/* continuation of procedure in Algorithm 3.2 from previous page*/
34: \/* Removal of non-executable interactions */
35: for each state \( s_j \in S \) do
36:   for each outgoing input transition \( t = s_j \xrightarrow{m(x)[g_j]} s_{j+1} \in \Gamma \) from state \( s_j \) do
37:     if exists no sequence of transitions \( p \) from the initial state \( s_0 \) to \( s_j \)
38:        such that \( p = s_0 \xrightarrow{\alpha_0[g_0]} s_1 \xrightarrow{\alpha_1[g_1]} \ldots s_{j-1} \xrightarrow{\alpha_{j-1}[g_{j-1}]} s_j \)
39:        in \( G_W \) such that there is a corresponding output transition \( t' = s_i \xrightarrow{b_0(x)[g_i]} s_{i+1} \in p \) with \( i < j \) then
40:          \( \Gamma \leftarrow \Gamma \backslash \{t\} \) \/* removes an the transition \( t \) from a set of transitions of \( G_W \) */
41:     end if
42:   end for
43: end for
44: end while
45: return \( G_W \)
46: end procedure
```

including all outgoing transitions (Lines 14-16). Finally, unreachable states and associated transitions are also eliminated in Line 17. On the other hand, in Lines 21-26 of the algorithm, if there is a controllable transition enabled at state \( p \) of \( G_W \) but not in \((p, q)\) of \( C^k \) then Line 24 of the algorithm marks this transition as disabled.

Algorithm 3.4 presents a minimization technique to deal with unsafe states and blocking states. This algorithm takes an SLTS \( C^k \) as its input. The system \( C^k \) is assumed to be the SLTS obtained after some iterations of the repeat until loop in Algorithm 3.1 (Lines 8-13). Specifically, \( C^k \) is the output of Algorithm 3.3. The iteration of the first for loop statement collects and stores all states from which no
Algorithm 3.3 Static Controllability Minimization

**Input:** $G_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ and $\mathcal{C}^k = (S_{\mathcal{C}^k}, S^0_{\mathcal{C}^k}, I_{\mathcal{C}^k}, O_{\mathcal{C}^k}, A_{\mathcal{C}^k}, \Gamma_{\mathcal{C}^k}, S_{\mathcal{F} \mathcal{C}^k}, V_{\mathcal{C}^k}, B_{\mathcal{C}^k})$

**Output:** Static controllable SLTS of $\mathcal{C}^k$, i.e., this algorithm produces an SLTS $\mathcal{C}^k$ such that all state of $G_W$ are state controllable with respect to $\mathcal{C}^k$

1: **procedure** staticControllability($G_W, \mathcal{C}^k$)
2: Let $S_{\mathcal{C}^k} = S^\text{Good}_{\mathcal{C}^k} \cup S^\text{Bad}_{\mathcal{C}^k}$, where $S^\text{Good}_{\mathcal{C}^k}$ is the set of safe states and $S^\text{Bad}_{\mathcal{C}^k}$ is the set of unsafe states
3: for each state $p \in S$ and a corresponding state $(p, q) \in S_{\mathcal{C}^k}$ do,
4:   for each static transition in $\Gamma$ with an event $t \in \Sigma_{uc}$ such that $t \in E_{S_{\mathcal{G}W}}(p)$ do
5:     if $t \not\in E_{S_{\mathcal{C}^k}}((p, q))$ then
6:         for all state $b \in S_{\mathcal{C}^k}$ with a dynamic transition $t' = b \xrightarrow{e(g)} (p, q)$ do
7:             Create a new state $s_{\text{Bad}}$ in $\mathcal{C}^k$ such that $z = b \xrightarrow{e(g)} s_{\text{Bad}}$
8:         end for
9:         $S^\text{Bad}_{\mathcal{C}^k} \leftarrow S^\text{Bad}_{\mathcal{C}^k} \cup \{s_{\text{Bad}}\}$
10:        $\Gamma_{\mathcal{C}^k} \leftarrow \Gamma_{\mathcal{C}^k} \cup \{z\}$
11:        end if
12:    end for
13:    Eliminate all transitions associated to the state $(p, q)$
14:    and update the set of transitions $\Gamma_{\mathcal{C}^k}$ accordingly
15:    $S_{\mathcal{C}^k} \leftarrow S_{\mathcal{C}^k}\backslash\{(p, q)\}$
16:    $\mathcal{C}^k \leftarrow \text{unreachableStateTransitionMinimization}(\mathcal{C}^k) \ \text{* remove unreachable states and associated transitions */}$
17:    end if
18: end for
19: for each static transition in $\Gamma$ with an event $t \in \Sigma_{uc}$ such that $t \in E_{S_{\mathcal{G}W}}(p)$ do
20:    if $t \not\in E_{S_{\mathcal{C}^k}}((p, q))$ then
21:        $E_{S_{\mathcal{G}W}}(p)\backslash\{t\} \ \text{/* disable t */}$
22:    end if
23: end for
24: return $\mathcal{C}^k$
25: end procedure
marked state is reachable or from which only bad states can be reached (Lines 8-12). The algorithm stores these states in the buffer $\text{BlockandUnsafe}$. For each state in $\text{BlockandUnsafe}$, Lines 14-19 of the algorithm create a new bad state $s^\text{Bad}$ and assign any dynamic transition that leads to a state in $\text{BlockandUnsafe}$ to $s^\text{Bad}$. This step is performed to preserve the structure of dynamic transitions, as done in Algorithm 3.3. Finally, Lines 22-28 eliminate all states and transitions collected in Lines 8-12. That is, all states in $\text{BlockandUnsafe}$ and all associated transitions that lead to a state in $\text{BlockandUnsafe}$ are removed.

Algorithm 3.5 presents an approach that can be used to compute a safe and dynamic controllable SLTS of a given system. This algorithm implements the second part of the definition of controllability given in Definition 3.3.8. It involves the generation and attachment of stronger guards to transitions and the collections of variables to be monitored at runtime as well as the removal of dynamic uncontrollable states and transitions. In addition, controllable dynamic transitions that lead to bad states are disabled.

First, we assume that the set of variables is partitioned into trackable and non-trackable variables. Trackable variables (which we will call “deterministic” variables because their occurrence is deterministic) are those variables whose values do not change from where they were declared to where they are being used, whereas non-trackable variables (which we will call nondeterministic variables because their occurrence is “nondeterministic”) are those whose values we cannot predict from when they were declared to when they are used. Specifically, trackable variables are associated with dynamic type 1 transitions while non-trackable variables are associated
Algorithm 3.4 Unsafe State Minimization

**Input:** $C_k = (S_{C_k}^0, S_{C_k}^F, I_{C_k}, O_{C_k}, A_{C_k}, \Gamma_{C_k}, S_{E_{C_k}}, V_{C_k}, B_{C_k})$

**Output:** nonblocking SLTS

1: procedure UnsafeStateMinimization($C_k$)

2: Let $S_{C_k} \leftarrow S_{C_k}^{\text{Good}} \cup S_{C_k}^{\text{Bad}}$, where $S_{C_k}^{\text{Good}}$ is the set of safe states and $S_{C_k}^{\text{Bad}}$ is the set of unsafe states

3: Let $\Sigma \leftarrow I_{C_k} \cup O_{C_k} \cup A_{C_k}$

4: Let $\text{BlockandUnsafe} = \emptyset$

5: /* The following for loop collects and store all the states that lead to blocking */

6: for each state $s \in S_{C_k}$ do,

7: if $\exists \delta \in \Sigma, \exists g \in B$ and $\exists s' \in S_{C_k}^{E} \setminus S_{C_k}^{\text{Bad}}$ such that $s \xrightarrow{\delta[g]} s'$ then

8: $\text{BlockandUnsafe} \leftarrow \text{BlockandUnsafe} \cup \{(s)\}$

9: end if

10: end for

11: /* The following for loop creates new bad states */

12: for all state $b \in S_{C_k} \setminus S_{C_k}^{\text{Bad}}$ with a dynamic transition such that $b \xrightarrow{e[g']} q$ and

13: $q \in \text{BlockandUnsafe}$ do

14: Create a new state $s_{\text{Bad}} \in S_{C_k}^{\text{Bad}}$ in $C_k$ such that $z = b \xrightarrow{e[g']} s_{\text{Bad}}$

15: $S_{C_k}^{\text{Bad}} \leftarrow S_{C_k} \cup \{s_{\text{Bad}}\}$

16: $\Gamma_{C_k} \leftarrow \Gamma_{C_k} \cup \{z\}$

17: end for

18: /* We eliminate all states and associated transitions in $\text{BlockandUnsafe}$ */

19: for all $q \in \text{BlockandUnsafe}$ do

20: for all $t \in \Gamma_{C_k}$, such that $t = q \xrightarrow{\alpha[g_1]}$ or $\exists b$ such that $t = b \xrightarrow{\alpha[g_2]} q$ do

21: /* eliminate all transitions associated with $q$ */

22: $\Gamma_{C_k} \leftarrow \Gamma_{C_k} \setminus \{t\}$

23: $S_{C_k} \leftarrow S_{C_k} \setminus \{q\}$

24: end for

25: end for

26: return $C_k$

27: end procedure
3.4. COMPOSITION SYNTHESIS ALGORITHM

with dynamic type 2 transitions. Non-trackable variables are the output of atomic operations. Now, the algorithm starts by collecting all transitions that lead to bad states from a given state (Lines 18-34). In these steps we keep track of transitions that lead to a bad state based on the evaluation of nondeterministic variables (Lines 19-25), this is given by the first if statement. The else statement after the if statement keeps track of transitions that lead to an unsafe state due to the evaluation of deterministic variables (Lines 26-31). The next step of the algorithm strengthens the guards of each transition (Lines 37-46). The value of deterministic variables can be tracked implies that we can trace back to where it was originally defined from where it is being used in order to strengthen the guard.

Given a transition which leads to a bad state due to deterministic variable \( z = s_x \xrightarrow{o[g_i(d)]} s_{x+1} \), we check every sequence of transitions of \( C^k \) to locate where it was declared first \( \xrightarrow{l_m(d)[g_j]} s_{j+1} \) and then we strengthen the guards which is given by taking the conjunction of the current guard on the transition and the negation of the guard of where it is being used \( \xrightarrow{t = s_j \xrightarrow{l_m(d)[g_j \land \neg g_i(d)]}} s_{j+1} \). Lines 49-61 of the algorithm check every state that has a transition that leads to a bad state due to nondeterministic variables for enforceable transitions. In the case that an enforceable transition is enabled at this state, we save the variable for runtime monitoring (Lines 51-53). The runtime monitoring involves equipping the generated controller with additional capability to be able to track a given variable for certain values and then trigger certain actions based on the values of this variable. On the other hand, if there is no enforceable transition enabled at this state, the algorithm (Lines 56-57) first creates a new state in \( \mathcal{S}_C^{Bad} \) and diverts all dynamic transitions to it as done
Algorithm 3.5 Dynamic Controllability and Guard Generation

Input: $G_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$ and $C^k = (S_{C^k}, S^0_{C^k}, I_{C^k}, O_{C^k}, A_{C^k}, \Gamma_{C^k}, S^F_{C^k}, V_{C^k}, B_{C^k})$

Output: A safe and dynamic controllable SLST of $C^k$ with respect to $G_W$

1: procedure dynamicControllabilityAndGuardGeneration($G_W, C^k$)  
2: Let $BP \leftarrow \emptyset$ be the set of bad state predicate transitions with trackable variables 
3: Let $BS \leftarrow \emptyset$ be the set of states from which a bad state is reachable based on trackable variables 
4: Let $BPN \leftarrow \emptyset$ be the set of bad state predicate transitions with non-trackable variables 
5: Let $BSN \leftarrow \emptyset$ be the set of states from which a bad state is reachable based on non-trackable variables 
6: Let $SP \leftarrow \emptyset$ be the set of safe state predicate 
7: Let $g_i(d)$ be a guard which depends on a variable $d \in V$ 
8: Let $Det \subseteq V$ be the set of deterministic variables 
9: Let $nonDet \subseteq V$ be the set of non deterministic variables (i.e., variables whose values depends on the output of atomic operations) 
10: Let $i \leftarrow 0, j \leftarrow 0$ 
11: Let $runtimeVariables$ be the set of variables to be monitored at runtime 
12: for all $s_x \in S_{C^k}$ do 
13: for each $\delta \in \Sigma_{uc}$ and $g_i(d) \in B$ enabled at state $s_x$ (i.e., $(\delta, g_i(d)) \in E^D_C(s_x)$) such that $t = s_x \xrightarrow{\delta[g_i(d)]} s_{x+1}$ in $C^k$ do, 
14: if $s_{x+1} \in S^B_{C^k}$ then 
15: if $d \in nonDet$ is the output variable of the atomic operation then 
16: $BPN \leftarrow BPN \cup \{t\}$ 
17: $BSN \leftarrow BSN \cup \{s_x\}$ 
18: end if 
19: else 
20: if $d \in Det$ then 
21: $BP \leftarrow BP \cup \{t\}$ 
22: $BS \leftarrow BS \cup \{s_x\}$ 
23: end if 
24: end if 
25: $i \leftarrow i + 1$ 
26: end for 
27: end for 
28: \/* continue with this procedure on next page*/
Algorithm 3.5 Dynamic Controllability and Guard Generation cont.

36: /* continuation of procedure in Algorithm 3.5 from previous page */
37: */ Guard propagation and Attachment */
38: for each $s_x \in BS$ do
39: for each $z \in BP$, such that $z = s_x \xrightarrow{\alpha [g_i(d)]} s_{x+1} \in BP$ do
40: for all path $r = s_0 \xrightarrow{\alpha_0 [g_0]} s_1 \xrightarrow{\alpha_1 [g_1]} \ldots s_j \xrightarrow{\alpha_j [g_j]} s_{j+1} \ldots s_x \xrightarrow{\alpha [g_i(d)]} s_{x+1} \ldots$ in $C^k$, $j < x$ do
41: $t \leftarrow s_j \xrightarrow{\lambda m(d)[g_i \land \neg g_i(d)]} s_{j+1}$ /* guard strengthening */
42: end for
43: end for
44: end for
45: end for
46: end for
47: */ Event Enforcement and collection of runtime monitoring variables */
48: for each $s_x \in BSN$ do
49: for each $z \in BPN$, such that $z = s_x \xrightarrow{\alpha [g_i(d)]} s_{x+1} \in BPN$ do
50: if exists $\alpha' \in E^D_{C^k}(s_x) \land \alpha' \in \Sigma_f$ then
51: save variable for runtime monitoring and enforcement
52: runtimeVariables $\leftarrow$ runtimeVariables $\cup \{d\}$ /* variable saved to be monitored at runtime */
53: else
54: Create new bad state $s^{Bad}$ and divert all dynamic transitions that lead to $s_x$ to $s^{Bad}$
55: Eliminate all transitions $(t, g) \in E^D_{C^k}(s_x)$
56: $S_{C^k} \leftarrow S_{C^k} \setminus \{s_x\}$
57: end if
58: end for
59: end for
60: $C \leftarrow$ disableControllableDynamicTransition($G_W, C$) /* disable dynamic controllable transitions */
61: return $C$
62: end procedure
3.4. COMPOSITION SYNTHESIS ALGORITHM

Algorithm 3.6 Disable Dynamic Controllable Transitions

Input: $G_W = (S, S^0, I, O, A, \Gamma, S^F, V, B)$, $C = (S_C, S_C^0, I_C, O_C, A_C, \Gamma_C, S_C^F, V_C, B_C)$

Output: $C$ where dynamic controllable transitions that do not satisfy system requirements are disabled.

1: procedure disableControllableDynamicTransition($G_W, C$)
2: for each state $p \in S$ such that $\exists \delta \in \Sigma_c$ and $\exists g \in B$ such that $(\delta, g) \in E^D_{g_w}(p)$ do
3: for each state $(p, q) \in S_C$ such that $\not\exists (\delta, g') \in E^D_{C \setminus S^B_C}(p, q)$
4: and $g \land g'$ is satisfiable do
5: $((\delta, (g \land \neg g')) \not\in E^D_{C \setminus S^B_C}(p, q)) \quad \text{* disable } (\delta, (g \land \neg g'))$, since $\delta \in \Sigma_c$ */
6: end for
7: end for
8: end procedure

in Algorithm 3.3. Next, we completely eliminate the entire state and all transitions associated with this state from $C^k$. Finally, Line 62 disables all dynamic controllable transitions (both dynamic type 1 and dynamic type 2 transitions) that are enabled at the plant state but not at the corresponding state of $C^k$.

Algorithm 3.6 is called by Algorithm 3.5 in Line 62 to disable dynamic controllable transitions that lead to a bad state. Once the iteration of the repeat until loop of Algorithm 3.1 has terminated, all states in $S^B_C$ would been made unreachable and there is no need to keep them. Hence, Algorithm 3.7 is called to remove all bad states and transitions in the set of bad states $S^B_C$. Algorithm 3.7 iterates over the set of states in $S^B_C$ and eliminates all states in $S^B_C$ including associated transitions.

Once all the issues relating to controllability and nonblocking have been dealt with, the next stage of the algorithm is to reverse the directions of the messages of the resulting controller. This step is done to allow for communication between
3.4. COMPOSITION SYNTHESIS ALGORITHM

Algorithm 3.7 Remove All Bad States and Associated Transition

**Input:** An SLTS $\mathcal{C}$ with states partitioned into good and bad states

**Output:** An SLTS $\mathcal{C}$ without bad states and transitions that lead to bad states

1: procedure removeUnsafeState($\mathcal{C}$)
2: \begin{algorithmic}
3: \While{$(\mathcal{S}_C^{\text{Bad}} \neq \emptyset)$}
4: \If{$(\exists t = s \xrightarrow{s_{\text{Bad}}} s_{\text{Bad}}$ such that $s_{\text{Bad}} \in \mathcal{S}_C^{\text{Bad}})$}
5: \Comment{eliminate all transitions associated with $s'$}
6: $\Gamma_C \leftarrow \Gamma_C \setminus \{t\}$
7: $\mathcal{S}_C \leftarrow \mathcal{S}_C \setminus \{s_{\text{Bad}}\}$
8: \EndIf
9: \EndWhile
10: return $\mathcal{C}$
11: end procedure

Algorithm 3.8 Reverse The Direction Of Messages Of $\mathcal{C}$

**Input:** An SLTS $\mathcal{C}$

**Output:** An SLTS $\mathcal{C}$ with input messages changed to output messages and vice versa.

1: procedure reverseMessageDirection($\mathcal{C}$)
2: \begin{algorithmic}
3: \For {each transition $t = s \xrightarrow{!m_{[g]}} s' \in \Gamma_c$ such that $!m \in \mathcal{I}_C$}
4: $t \leftarrow s \xrightarrow{?m_{[g]}} s' \in \Gamma_c$ \Comment{set output messages to input messages}
5: $\mathcal{I}_C \leftarrow \mathcal{I}_C \cup \{?m\}$
6: $\mathcal{O}_C \leftarrow \mathcal{O}_C \setminus \{!m\}$
7: \EndFor
8: \For {each transition $t = s \xrightarrow{?m_{[g]}} s' \in \Gamma_c$ such that $?m \in \mathcal{O}_C$}
9: $t \leftarrow s \xrightarrow{!m_{[g]}} s' \in \Gamma_c$ \Comment{set output messages to input messages}
10: $\mathcal{O}_C \leftarrow \mathcal{O}_C \cup \{!m\}$
11: $\mathcal{I}_C \leftarrow \mathcal{I}_C \setminus \{?m\}$
12: \EndFor
13: return $\mathcal{C}$
14: end procedure
the plant and the controller. This ensures that an output message in the plant’s transition system can be consumed by an input message in the controller’s transition system and vice versa. Given an SLTS $C$ as the input to Algorithm 3.8, it reverses the direction of the messages of $C$. That is, given an input (respectively, output) message $?m(x)$, Algorithm 3.8 will change it to an output (respectively, input) message $!m(x)$.

A composition generation process may fail if the given plant cannot simulate its specification. Algorithm 3.9 iteratively pares down a given specification so that it can be simulated by a given plant. The algorithm takes $\mathcal{G}_W$ and $\mathcal{T}_W$ as inputs such that there is no simulation relation between $\mathcal{G}_W$ and $\mathcal{T}_W$ and returns a new specification $\mathcal{T}_W'$ that can be simulated by $\mathcal{G}_W$. Line 2 of Algorithm 3.9 defines a maximal relation $R$ given by the Cartesian product of the states of $\mathcal{G}_W$ and $\mathcal{T}_W$.

Now, Line 6 of the algorithm iterates over each pair of reachable states $(t_i, s_i) \in R$ such that there exists a transition $t_i \xrightarrow{\delta[g_{i_1}]} t'_i \in \Gamma_{\mathcal{T}_W}$ and then checks for the following three cases where a plant may fail to simulate a given specification.

- There is no matching transition at state $s_i$ of the plant. In this case the transition $t_i \xrightarrow{\delta[g_{i_1}]} t'_i \in \Gamma_{\mathcal{T}_W}$ will be removed (Lines 9-11).

- There exists a transition $s_i \xrightarrow{\delta[g_{i_2}]} s'_i \in \Gamma_{\mathcal{G}_W}$ and $(t_i, s_i) \in R$ but the guard $g_{i_1}$ is not a subguard of $g_{i_2}$, i.e., $g_{i_1} \not\leq g_{i_2}$. In this case the guard on $\mathcal{T}_W$ is strengthened to that of $\mathcal{G}_W$ (Lines 13-16).

- There exists a transition $s_i \xrightarrow{\delta[g_{i_2}]} s'_i \in \Gamma_{\mathcal{G}_W}$ and $g_{i_1} \leq g_{i_2}$ but $(t_i, s_i) \notin R$. In this case the transition $t_i \xrightarrow{\delta[g_{i_1}]} t'_i \in \Gamma_{\mathcal{T}_W}$ is eliminated (Lines 16-21).
The algorithm terminates when all transitions that prevent the specification from being simulated by the plant have been dealt with, i.e., $T^W \equiv T^{W'}$. Algorithm 3.10

**Algorithm 3.9 Refine Target To Be Simulated By Plant**

**Input:** $G_W = (S_{g_W}, S_{g_W}^0, I_{g_W}, O_{g_W}, A_{g_W}, \Gamma_{g_W}, \mathcal{F}_{g_W}, V_{g_W}, \mathcal{B}_{g_W})$ and $T^W = (S_{T^W}, S_{T^W}^0, I_{T^W}, O_{T^W}, A_{T^W}, \Gamma_{T^W}, \mathcal{F}_{T^W}, V_{T^W}, \mathcal{B}_{T^W})$

**Output:** $T^{W'}$ where $T^{W'}$ is derived from $T^W$ and $T^{W'} \preceq G_W$

1: procedure refineTargetToBeSimulatedByPlant($T^W, G_W$)
2:   Let $R \leftarrow S_{T^W} \times S_{g_W}$
3:   Let $T^{W'} \leftarrow \emptyset$
4: repeat
5:   $T^{W'} \leftarrow T^W$
6:   for each $(t_i, s_i) \in R$ such that there exist sequences of transitions $t_0 \xrightarrow{\delta_0[g_i]} t_1 \xrightarrow{\delta_1[g_i]} t_2 \ldots t_{i-1} \xrightarrow{\delta_{i-1}[g_i]} t_i, t_0 \in S^0_{T^W}$ and $s_0 \xrightarrow{\delta_0[g_i]} s_1 \xrightarrow{\delta_1[g_i]} s_2 \ldots s_{i-1} \xrightarrow{\delta_{i-1}[g_i]} s_i, s_0 \in S^0_{g_W}$ do
7:     if there is a transition $t_i \xrightarrow{\delta_{i}[g_i]} t'_i \in \Gamma_{T^W}$ then
8:       if (there is no transition $s_i \xrightarrow{\delta_{i}[g_i]} s'_i \in \Gamma_{g_W}$ such that $s_i$, $s'_i \in S^0_{T^W}$) then
9:         $\Gamma_{T^W} \backslash \{t_i \xrightarrow{\delta_{i}[g_i]} t'_i\}$ /* this removes the sequence of transition from the initial state to $t'_i$ in $T^W$ */
10:        else if (there is a transition $s_i \xrightarrow{\delta_{i}[g_i]} s'_i \in \Gamma_{g_W}$ and $(t'_i, s'_i) \in R$ but $g_i \leq g_{i_2}$) then
11:           $g_i \leftarrow g_{i_2}$ /* change the guards to that of $G_W$ */
12:        end if
13:     end if
14:     else if (there is a transition $s_i \xrightarrow{\delta_{i}[g_i]} s'_i \in \Gamma_{g_W}$ and $(g_i, g_{i_2}) \in R$) then
15:           but $(t'_i, s'_i) \notin R$ then
16:             $\Gamma_{T^W} \backslash \{t_i \xrightarrow{\delta_{i}[g_i]} t'_i\}$ /* this removes the sequence of transitions from the initial state to $t'_i$ in $T^W$ */
17:         end if
18: end for
19: until $(T^{W'} \equiv T^W)$
20: return $T^{W'}$
21: end procedure
3.4. COMPOSITION SYNTHESIS ALGORITHM

presents a procedure to modify a given SLTS to include a special transition (\(\epsilon\) transition) which we call a “timeout” transition which will be used at runtime to provide the generated controller the ability to preempt certain dynamic type 2 transitions (which we call “preemptable transitions”) using enforceable transitions. Self-loops are treated in a similar way. The technique presented here is identical to that in the work by Wonham and Cai [161].

Algorithm 3.10 Plant and Specification Adjustment

**Input:** An SLTS \(\mathcal{M} = (S_M, S^0_M, \mathcal{T}_M, O_M, A_M, \Gamma_M, S^F_M, V_M, B_M)\)

**Output:** A new SLTS constructed by modifying \(\mathcal{M}\) to include a “timeout” transition (\(\epsilon\) transition) to enable event enforcement at runtime.

```plaintext
1: procedure plantAndSpecAdjustment(\(\mathcal{M}\))
2:     initialize the set dynamic_2_Transition to be all dynamic type 2 transitions
3:     such that dynamic_2_Transition \(\subseteq \Gamma_M\)
4:     /* initialize the set of dynamic 2 transitions which forms the set of preemptable transitions*/
5:     for each state \(s \in S_M\) do
6:         if \((\exists \delta \in \Sigma_f\) and exists a guard \(g\) such that \(s \xrightarrow{\delta | g} \in \Gamma_M\)) \(\wedge (\exists \lambda\) and exists a guard \(g'\) such that \(s \xrightarrow{\lambda | g'} \in\) dynamic_2_Transition) then
7:             split \(s\) into 2 states \(s'\) and \(s''\) and create a new transition such that \(s' \xrightarrow{\epsilon} s''\) where \(\epsilon\) serves as a delay to help preempt preemptable events.
8:             for each transition \(t\) enabled at \(s\) do
9:                 if \(t\) in dynamic_2_Transition then
10:                     define \(t\) with its source state at \(s''\)
11:                 else define \(t\) with its source state at \(s'\)
12:             end if
13:         end for
14:     end if
15:     \((S_M \cup \{s'\} \cup \{s''\}) \backslash \{s\}\)
16:     \(\Gamma_M \cup \{s' \xrightarrow{\lambda} s''\}\)
17: end for
18: return \(\mathcal{M}\)
19: end procedure
```
Example 3.4.1 below is a small example to illustrate how Algorithm 3.1 works.

Example 3.4.1.

Consider the plant $G_W$ (assumed to be the asynchronous parallel composition of some given services) in Figure 3.10(a) and the specification $T^W$ in Figure 3.10(b) as the inputs to Algorithm 3.1. As noted above, input and function invocation transitions are considered as uncontrollable and output transitions are controllable (i.e., $!msg_1(x), !msg_3(var), !msg_2(z) \in \Sigma_c, ?msg_2(z), ?msg_1(x), ?msg_3(var), atom_3(x), atom_1(x :: y), atom_2(y :: v), atom_3(x), ?fail() \in \Sigma_{uc}, !_fail() \in \Sigma_f$). Clearly, $G_W$ simulates $T^W$ and both SLTSs are communication-error free. Line 6 of Algorithm 3.1 computes the composition refinement $C^0$ which is represented in Figure 3.10(c). Applying the rest of Algorithm 3.1(Lines 8-26) to $C^0$ gives the following (where we will only consider steps of the algorithm that are relevant to this example):

1. (Algorithm 3.1 Lines 8-13) within the repeat until loop we have the following steps:

   (a) $k=1$ means we will pass $C^0$ to the three functions within the repeat until loop (Algorithm 3.1 Lines 8-13).

   (i) **Static Controllability** (Line 10 Algorithm 3.1)

   Starting from the initial state of $C^0$ every state of $C^0$ satisfies the static controllability condition except for state $(s_5, t_5)$ where there is an uncontrollable static transition $(s_5) \xrightarrow{?msg_2(z)} (s_6)$ enabled in the plant but not at the corresponding state $(s_5, t_5)$ of $C^0$. This implies that state $(s_5, t_5)$ and the associated transitions $((s_5, t_5) \xrightarrow{atom_3(x)} (s_2, t_2)$
and \((s_3, t_3) \xrightarrow{?msg_1(x)} (s_5, t_5)\) will be eliminated from \(C^0\) (Lines 14-17 of Algorithm 3.3). This step produces the SLTS in Figure 3.10(d).

(ii) **Blocking states** (Line 11 Algorithm 3.1)

At this stage no state of the \(C^0\) in Figure 3.10(d) is blocking, we proceed to the next step.

(iii) **Dynamic Controllability** (Line 12 Algorithm 3.1)

At this stage of the algorithm, there are three transitions that lead to bad states \(((s_2, t_2) \xrightarrow{atom_1(x::y)[3 \leq x < 7]} (BAD_1), (s_4, t_4) \xrightarrow{atom_2(y::v)[y = a]} (BAD_2), (s_1, t_1) \xrightarrow{?msg_3(var)[var \in \{db\}]} (BAD_3))\) in the \(C^0\) of Figure 3.10(d).

By applying the next step of the algorithm we have the following.

- **Strengthening and Attachment of Guards** (Lines 38-46 of Algorithm 3.5)

  The two uncontrollable transitions \((s_2, t_2) \xrightarrow{atom_1(x::y)[3 \leq x < 7]} (BAD_1)\) and \((s_1, t_1) \xrightarrow{?msg_3(var)[var \in \{db\}]} (BAD_3)\) are dynamic type 1 transitions since the values of the variables \(x\) and \(var\) can be tracked from where they were declared. Hence, at this stage the algorithm strengthens the guards of these transitions so that the state \(BAD_1\) and \(BAD_3\) are made unreachable by attaching the guard \(\neg(3 \leq x < 7)\) and \(var \notin \{db\}\) to the transition \((s_0, t_0) \xrightarrow{!msg_1(x)} (s_1, t_1)\) and the transition \((s_0, t_0) \xrightarrow{!msg_3(var)} (s_1, t_1)\), respectively. The resultant SLTS is given in Figure 3.10(e).

- **Event Enforcement and Collection of Runtime Variable** (Lines 49-61 of Algorithm 3.5)
3.4. COMPOSITION SYNTHESIS ALGORITHM

The transition \((s_4, t_4) \xrightarrow{\text{atom}_2(y::v)[y=a]} (BAD_2)\) is a dynamic type 2 transition since the variable \(y\) depends on the output of the atomic operation \(\text{atom}_1(x :: y)\) which cannot be tracked. This step of the algorithm will check for the enforceable transition \((s_4, t_4) \xrightarrow{!\text{fail}()} (s_0, t_0)\) and save the variable \(y\) to be monitored at runtime and then enforce \(!\text{fail}()\) when \(y=a\), to prevent \(BAD_2\) from being reached.

(b) \(k=2\) means we will pass \(C^1\) (Figure 3.10(e)) to the three functions in the \texttt{repeat until} loop (Algorithm 3.1 Lines 8-13)

(i) **Static Controllability** (Line 10 Algorithm 3.1)

Again starting from the initial state of \(C^1\) (Figure 3.10(e)) every state of \(C^1\) satisfies the static controllability condition except for state \((s_3, t_3)\) where there is an uncontrollable static transitions \((s_3) \xrightarrow{?msg_1(x)} (s_5)\) enabled in the corresponding state of the plant but not at the state \((s_3, t_3)\) of \(C^1\). This implies that state \((s_3, t_3)\) and the associated transitions \((s_3, t_3) \xrightarrow{?msg_2(z)} (s_1, t_1)\) and \((s_7, t_7) \xrightarrow{!msg_2(z)} (s_3, t_3)\) will be eliminated from \(C^1\) (Lines 14-17 of Algorithm 3.3).

(ii) **Blocking states** (Line 11 Algorithm 3.1)

Now, because in the previous step the transition from state \((s_7, t_7)\) to state \((s_3, t_3)\) was eliminated, state \((s_7, t_7)\) becomes a blocking state since it is not a final state and does not lead to a final state. Given this we have the following steps:
create a new state $BAD_4$ and assign any dynamic transition leading to state $(s_7, t_7)$ to $BAD_4$ (Algorithm 3.4 Lines 16-18)

then eliminate $(s_7, t_7)$ (Algorithm 3.4 Lines 22-23)

The results of (i) and (ii) are shown in the diagram in Figure 3.10(f).

(iii) Dynamic Controllability (Line 12 Algorithm 3.1)

A new bad state $BAD_4$ was added to $C^1$ and needs to be made unreachable. (Note that all other BAD states are still unreachable.) We have the following step:

- **Strengthening and Attachment of Guards** (Lines 38-46 of Algorithm 3.5)

$(s_1, t_1) \xrightarrow{?msg_3(var)[var \in \{cr\}]} (BAD_4)$ is an uncontrollable dynamic type 1 transitions since the values of the variable $var$ can be tracked from when it was declared. Hence, at this stage the algorithm strengthens the guard of this transition so that the state $BAD_4$ is made unreachable by attaching the guard $var \notin \{cr\}$ to the transition $(s_0, t_0) \xrightarrow{msg_3(var)[var \notin \{db\}]} (s_1, t_1)$ (shown Figure 3.10(f)).

(c) $k=3$ means we will pass $C^2$ (Figure 3.10(f)) to the three functions in the repeat until loop.

Iterating over the repeat until loop again will return the same SLTS shown in Figure 3.10(f), i.e., $C^3 = C^2$, hence the loop terminates.

2. **Line 14 of Algorithm 3.1**
Now, once the loop terminates, the next stage is to remove all the bad states 
\((BAD_1, BAD_2, BAD_3, BAD_4)\) from \(C^3\) by calling Algorithm 3.7. In addition, 
all transitions in and out of these bad states are removed.

3. **Line 16 of Algorithm 3.1**

\(C^3\) is not communication-error free since the transition \((s_0, t_0) \xrightarrow{msg_3(var)[var \notin \{db,cr\}]} (s_1, t_1)\) can emit the message \(!msg_3(var)\) but there is no matching input transition in \(C^3\) to consume it, hence this step eliminates \((s_0, t_0) \xrightarrow{msg_3(var)[var \notin \{db,cr\}]} (s_1, t_1)\) from \(C^3\). Next, the \texttt{if} statement in Line 15 requires that the execution 
goes back to Line 7, but in this example when we go back to Line 7 the result 
remains the same.

4. **Line 19 of Algorithm 3.1**

This step reverses the messages of \(C^2\) producing the final output \(C\) as shown in 
Figure 3.10(g).

3.4.1 **Discussion: Manual Steps and Consequence of Design Choices of 
Forcible Events**

All the stages of the methodology presented in Figure 3.9 are fully automated. However, the designer/modeler is expected to have some basic knowledge of DES and the 
service composition, since the initial construction of the plant and the specification is 
done manually. Apart from specifying the plant and specification, there is only one 
manual step at design time that the designer/modeler must consider. The designer 
must determine how to add forcible transitions at the right places of both the plant
and the specification. The task of labeling an existing transition as enforceable transition or creating a new enforceable transition in the controlled system to be used to preempt dynamic type 2 transitions is mostly dependent on the domain and the designer perspective. In addition, choosing the target state of an enforceable transition is dependent of the current state of the system and the domain being modeled. That is, in case there is a failure as a result of an unsuspected output of a dynamic type 2 transition, the designer must determine what state should the system go to, whether the system should transition to the initial state, and whether the system should try to do the previous transition again. In Example 3.4.1 choosing the enforceable transition was easy since the plant already has a transition \((s_4, t_4) \xrightarrow{\text{fail}(\cdot)} (s_0, t_0)\) that leads to the initial state and since this example has no domain restriction. Finally, it is possible to have two or more forcible transitions outgoing a given state \(s\), to ensure that we achieve a comparable minimally restrict solution, each forcible transitions will be assigned to a given set of transitions that it can preempt at runtime. Thus, two forcible transitions cannot be used to preempt the same transition at a given state \(s\).
Figure 3.10: Illustrative Example Using the Algorithm
3.4. COMPOSITION SYNTHESIS ALGORITHM

(c) Composition refinement $C^0 = \mathcal{G}_W \times_{ref} \mathcal{T}^W$

Figure 3.10: Illustrative Example Using the Algorithm
(d) $C^0$ after applying static controllability

Figure 3.10: Illustrative Example Using the Algorithm
(e) $C^1$ (the variable $y = a$ is monitored at runtime)

Figure 3.10: Illustrative Example Using the Algorithm
3.4. COMPOSITION SYNTHESIS ALGORITHM

Figure 3.10: Illustrative Example Using the Algorithm
We have applied Algorithm 3.1 to various small examples. Also, Algorithm 3.1 has been manually applied to the flight booking example introduced earlier. The asynchronous parallel composition forming the plant of this example has 150 states and 3410 transitions while computing the composition refinement of the plant and the specification yielded a transition system $C^0$ with 175 states and 3945 transitions. Applying Lines 6-16 of the algorithm to $C^0$ will further reduce the number of states and transitions.
3.5 Proof of Correctness and Completeness

In this section, we present a theorem and a proof that proves the correctness of our approach. Theorem 3.5.1 shows that there exists a controller which solves the composition problem stated in Definition 3.3.11. We will start with various definitions and then we will state the main theorem of the section and finally provide a constructive proof for the theorem.

3.5.1 Proof of Controller Existences

Before we prove Theorem 3.5.1, let us consider the following lemma resulting from observations made from the construction of the controller $C$ by Algorithm 3.1. Let $S_C$ and $S_{C^0}$ denote the set of states of $C$ and $C^0$, respectively. Also, let $S_{C^0} \setminus S^{Bad}_{C^0} = \{ s \mid s \in S_{C^0} \land s \notin S^{Bad}_{C^0} \}$. In the proofs that follow, we will refer to bad states $S^{Bad}_C$ as states that are not reachable in the specification or that violate controllability or nonblocking conditions. Also, during the construction of $C$ by Algorithm 3.1, new states are created. These states are also marked as bad states, since they do not satisfy either some controllability or nonblocking conditions.

The first lemma says that in constructing $C$, our algorithm only removes states from $C^0$ that are bad. That is, a state in the resulting $C$ is a state in the original $C^0$ and is not a bad state.

**Lemma 3.5.1.** Given a controller $C$ generated by Algorithm 3.1 such that $C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W$, then $S_C \subseteq S_{C^0} \setminus S^{Bad}_{C^0}$.

**Proof.** The proof is done constructively.
In Algorithm 3.1, Line 6, \( \mathcal{C} \) is initially given by \( \mathcal{C}^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}^W \) and \( \mathcal{S}_{\mathcal{C}^0} = \mathcal{S}_{\mathcal{C}^0}^{\text{Good}} \cup \mathcal{S}_{\mathcal{C}^0}^{\text{Bad}} \). This implies that at this stage in the construction of \( \mathcal{C} \) by the algorithm, the set of states of \( \mathcal{C} \) is equal to the set of states of \( \mathcal{C}^0 \). That is, \( \mathcal{S}_\mathcal{C} = \mathcal{S}_{\mathcal{C}^0} \) which means \( \mathcal{S}_\mathcal{C} = \mathcal{S}_{\mathcal{C}^0}^{\text{Good}} \cup \mathcal{S}_{\mathcal{C}^0}^{\text{Bad}} \). Let \( \mathcal{C}^k \) denote the resultant SLTS obtained after some \( k \) iterations of the \texttt{repeat until} loop of Algorithm 3.1 on \( \mathcal{C}^0 \). Now, to show that \( \mathcal{S}_\mathcal{C} \subseteq \mathcal{S}_{\mathcal{C}^0} \setminus \mathcal{S}_{\mathcal{C}^0}^{\text{Bad}} \), we prove the following:

(i) Upon termination of Algorithm 3.1, all the states in \( \mathcal{S}_{\mathcal{C}^0}^{\text{Bad}} \) have been made unreachable and eliminated and hence would not be in the final output \( \mathcal{C} \) of Algorithm 3.1,

(ii) In the iterations of Algorithm 3.1, some of the states in \( \mathcal{S}_{\mathcal{C}^0}^{\text{Good}} \) become bad states and are made unreachable and eliminated,

(iii) All new bad states created by the algorithm are made unreachable and eliminated before the algorithm terminates.

To show (i), let \( q \in \mathcal{S}_{\mathcal{C}^0}^{\text{Bad}} \) and suppose that \( \exists p \in \mathcal{S}_{\mathcal{C}^0}, \exists \delta \in \Sigma \) and \( \exists g \in \mathcal{B} \) such that the transition \( t = p \overset{\delta[g]}{\longrightarrow} q \) in \( \mathcal{C}^0 \), then in the first iteration of the \texttt{repeat until} loop of Algorithm 3.1 one of the following holds:

(a) if \( \delta \in \Sigma_{\text{uc}} \) then

- in the case that \( t \) is static, Lines 14-17 of Algorithm 3.3 will eliminate \( t \) which implies that \( q \) is also eliminated.
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

• in the case that $t$ is a dynamic type 1 transition, then the guard $g$ will be strengthened by Algorithm 3.5 in Lines 38-46. Hence, $q$ becomes unreachable from any good state and finally deleted at Line 14 of Algorithm 3.1.

• in the case that $t$ is a dynamic type 2 transition, then by Corollary 3.3.1, Algorithm 3.5 Lines 49-61 will ensure that there is an enforceable transition also enabled at state $p$ to preempt $t$ at runtime. Hence, state $q$ becomes unreachable and is later deleted at Line 14 of Algorithm 3.1.

(b) if $\delta \in \Sigma_c$ then

in all cases $t$ (static or dynamic transition) would be disabled by Algorithm 3.1, making $q$ unreachable and later deleted at Line 14 of Algorithm 3.1.

To show (ii) we note that after some $k$ iterations of the repeat until loop of Algorithm 3.1 on $C^0$ some good state in $S^Good_c$ becomes bad due to controllability (Definition 3.3.8) or blocking (Definition 3.3.9). These new bad states are treated in the same way as done in (i), which implies that they are never reachable in the final output of Algorithm 3.1.

The proof of (iii) is as follows: During the construction of $C$, Algorithm 3.1 creates completely new bad states in the process of constructing $C$ (Lines 9–12 of Algorithm 3.3, Lines 16–18 of Algorithm 3.4 and Line 56 of Algorithm 3.5). However, these new bad states are also treated and deleted in the same way as in (i) before the termination of Algorithm 3.1.

From (i), (ii) and (iii) it is clear that by the time Algorithm 3.1 terminates the set of bad states of its final output $C$ will be empty, i.e., $S^Bad_C = \emptyset$, and some of
the states in $S_{C_0}^{Good}$ may have been converted into bad states and removed too. This means that the set of states of $C$ is only a subset of $S_{C_0}^{Good}$. Thus, $S_C \subseteq S_{C_0} \setminus S_{C_0}^{Bad}$. ■

In the following lemma we show that a state $s \in S_{C_0} \setminus S_C$ is either a bad state in $C^0$ or for some $k \in \mathbb{N}$, $s$ was made bad at the $k^{th}$ iteration of the repeat until loop of Algorithm 3.1 over $C^0$.

**Lemma 3.5.2.** Given that $C^0 = G_W \times_{ref} T^W$ and $s \in S_{C_0} \setminus S_C$, then the following holds:

(i) $s \in S_{C_0}^{Bad}$,

(ii) $s \in S_{C_k}^{Bad}$ where $C^k$ is the SLTS obtained after some $k$ iterations of the repeat until loop of Algorithm 3.1

**Proof.** Proof of (i): given that $s \in S_{C_0} \setminus S_C$ implies that $s \notin S_C$ and from Lemma 3.5.1 it implies that $s \in S_{C_0}^{Bad}$ which proves (i).

Proof of (ii): This follows from Lemma 3.5.1 item (ii) by noting that, during the iterations of the repeat until loop of Algorithm 3.1, a good state $s$ in $C^{k-1}$ is changed to a bad state $s \in S_{C_k}^{Bad}$ because either $s$ is not state controllable or leads to a violation of controllability (Definition 3.3.8) or results in blocking (Definition 3.3.9). ■

In the following lemma we show that the set of good states of $C$ is a subset of the set of good states of $C^0$.

**Lemma 3.5.3.** Given a controller $C$ generated by Algorithm 3.1 such that $C^0 = G_W \times_{ref} T^W$, then $S_C^{Good} \subseteq S_{C_0}^{Good}$.
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

Proof. The proof of this lemma is similar to that of Lemma 3.5.1 and so we omit it here.

Theorem 3.5.1 (Controller Existence). Given a system modeled by an SLTS $G_W$ and a specification $T^W$ with $T^W \preceq G_W$, a controller $C$ exists such that $C \otimes G_W \preceq T^W$ if and only if $G_W$ is state controllable with respect to $T^W$.

Proof. The proof is done constructively and in two parts. We denote each part with gray boxes later in the proof.

Let $G_W = (S_{G_W}, S^0_{G_W}, I_{G_W}, O_{G_W}, A_{G_W}, \Gamma_{G_W}, S^F_{G_W}, V_{G_W}, B_{G_W})$ and $T^W = (S_{T^W}, S^0_{T^W}, I_{T^W}, O_{T^W}, A_{T^W}, \Gamma_{T^W}, S^F_{T^W}, V_{T^W}, B_{T^W})$. Let $Composer(G_W, T^W)$ denote the transition system obtained from $G_W$ and $T^W$ by applying Algorithm 3.1 (i.e., the final output of Algorithm 3.1). In addition, let $C = Composer(G_W, T^W)$.

We will denote the set of states of $C^0$ as $S_{C^0}$ and the set of transitions of $C^0$ as $\Gamma_{C^0}$. Let $S^\text{Good}_{C^0}$ and $S^\text{Bad}_{C^0}$ denote the set of good states of $C^0$ and the set of bad states of $C^0$, respectively. Let $C^0 \setminus S^\text{Bad}_{C^0}$ denote the SLTS obtained after removing the set of bad states $S^\text{Bad}_{C^0}$ from $C^0$, i.e., $C^0$ excluding the set of bad states. We will assume similar notation for $C$. We make the following observations. In Algorithm 3.1, $C$ is initially computed from $G_W \times_{\text{ref}} T^W$ (Line 6) and then certain reduction steps are further performed on it. By definition, $C^0 = G_W \times_{\text{ref}} T^W$ in Line 6 of Algorithm 3.1 (definition of composition refinement). The states of $C^0$ are partitioned into the set of good states and the set of bad states, respectively. That is, $S_{C^0} = S^\text{Good}_{C^0} \cup S^\text{Bad}_{C^0}$. The set of good states and transitions leading to good states of $C^0$ lie in $T^W$. Specifically, $C$ is obtained from $C^0$ after removing certain transitions and bad states.

The first part of the proof of this theorem entails showing that given a controller $C$
generated by Algorithm 3.1 and \( G_W \) controllable with respect to \( T^W \), then \( C \otimes G_W \preceq T^W \) holds. This is presented formally in the box below.

Let \( C \otimes G_W = (S_C \otimes G_W, S_0^C \otimes G_W, I_C \otimes G_W, O_C \otimes G_W, A_C \otimes G_W, \Gamma_C \otimes G_W, S_F \otimes G_W, V_C \otimes G_W, B_C \otimes G_W) \) denote the controlled system when \( G_W \) is under control of \( C \).

**Part 1:**

if: Assume that \( C = \text{Composer}(G_W, T^W) \) and \( T^W \) is controllable.

To prove:

1. \( C \otimes G_W \preceq T^W \)

That is, show that when the plant is coupled with \( C \), the resultant transition system is simulated by \( T^W \).

**Example 3.5.1.** First let us illustrate this part of the proof of Theorem 3.5.1 by an example (i.e., given that \( C = \text{Composer}(G_W, T^W) \) and \( T^W \) is controllable, show that \( C \otimes G_W \preceq T^W \)). Let the SLTS in Figure 3.10(a) and Figure 3.10(b) represent the plant and the specification, respectively. It can be verified that \( T^W \preceq G_W \). Figure 3.10(c) is the result of computing the composition refinement \( (C^0 = G_W \times_{\text{ref}} T^W) \) of the SLTSs in Figure 3.10(a) and Figure 3.10(b), respectively. Now, it can be seen that \( C^0 \) is made up of good states and bad states. Our proof to the theorem will be in two parts. The first part of the proof would be to show that \( (C^0 \otimes G_W) \) simulates \( T^W \) when the transitions \((s_2, t_2) \xrightarrow{\text{atom}_1(x::y)[3 \leq x < 7]} (BAD_1), (s_4, t_4) \xrightarrow{\text{atom}_2(y::v)[y=a]} (BAD_2), (s_1, t_1) \xrightarrow{\text{msg}_3[\text{var}]} (BAD_3)\) have been eliminated from \( C^0 \) and the bad states made unreachable. It can be seen that the controller \( C \) in Figure 3.10(g) has no transitions that lead to bad states and it can also be easily verified that \( C \otimes G_W \preceq T^W \).
which will constitute the proof of the second part. That is, the second part of the
proof would be to show that \( C \) generated from \( C^0 \) by Algorithm 3.1 has no bad states
and has no transitions that lead to bad states and, hence, satisfies \( C \otimes G_W \preceq T^W \).

The proof of Part 1 will proceed as follows. Firstly, in (a) we define a relation
\( R \) and show that \( R \) is a simulation relation between \( (C^0 \otimes G_W) \) and \( T^W \) when \( C^0 \)
is restricted to having only good states, i.e., \( (C^0 \setminus S^\text{Bad}_{C^0} \otimes G_W) \preceq T^W \). Secondly, in
(b) we shall further define another relation \( R \upharpoonright S_C \), a subset of \( R \) and establish that
\( R \upharpoonright S_C \) is a simulation relation between \( (C \otimes G_W) \) and \( T^W \) if \( C \) is generated without
bad states and transitions that lead to bad states. Finally, in (c) we shall show that
actually \( C \) has no bad states and has no transitions that lead to bad states.

We define the relation
\[
R = \{ (((s_n, t_n), s_n), t_n) \mid \exists \delta_0, \delta_1, \ldots, \delta_n \in \Sigma, \exists s_0, s_1, \ldots, s_n \in S_{G_W}, \exists t_0, t_1, \ldots, t_n \in S_{T_W},
\exists g_0, g_1, \ldots, g_n \in B:
(s_0 \in S^0_{G_W}, t_0 \in S^0_{T_W}, [\forall (0 < i \leq n) [s_{i-1} \xrightarrow{\delta_{i-1} [g_{i-1}]} s_i \in \Gamma_{G_W}] \land [t_{i-1} \xrightarrow{\delta_{i-1} [g_{i-1}]} t_i \in \Gamma_{T_W}]]) \}
\]
\[(3.1)\]

(a) We will start by showing that \( R \) is a simulation relation between \( C^0 \otimes G_W \) and
\( T^W \) when the states of \( C^0 \otimes G_W \) are constrained to only good states, i.e., \( T^W \)
simulates \( ((C^0 \setminus S^\text{Bad}_{C^0}) \otimes G_W) \).

Consider \( (((s_n, t_n), s_n), t_n) \in R \) and \( (s_n, t_n) \in S^\text{Good}_{C^0} \),
and suppose \( ((s_n, t_n), s_n) \xrightarrow{\delta_n [g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C^0 \otimes G_W} \) for some \( \delta_n \in \Sigma \)
and \( (s_{n+1}, t_{n+1}) \in S^\text{Good}_{C^0} \), then by definition of \( \times_{\text{ref}} \) used in the construction of
\[ C^0 \text{ it implies that } \exists \ t_{n+1} \in S_T \text{ such that } t_n \xrightarrow{\delta_n(y_n)} t_{n+1} \in \Gamma_T \text{ and therefore } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R. \]

This is because from the definition of \( C^0 = G_W \times_{\text{ref}} T^W \), every transition that leads to a good state in \( C^0 \) from \( (s_n, t_n) \) can be matched by \( T^W \). This implies that \( R \) is a simulation relation between \((C^0 \backslash S_{C^0}^{\text{Bad}}) \otimes G_W) \) and \( T^W \).

As in Example 3.5.1 if we eliminate all bad states and their associated transitions

\[
(s_2, t_2) \xrightarrow{\text{atom}_1(x:y)[3 \leq x < 7]} (BAD_1), (s_4, t_4) \xrightarrow{\text{atom}_2(y:x)[y=a]} (BAD_2),
\]

\[
(s_1, t_1) \xrightarrow{\text{?msg}(\text{var})[\text{var} \in \{db\}]} (BAD_3)
\]

from \( C^0 \) of Figure 3.10(c) and combine it with plant in Figure 3.10(a) we get a system that is simulated by the specification in Figure 3.10(b).

Now, in (b) and (c) we prove the following claim:

\[ C \otimes G_W \preceq T^W \]

(b) (Transitions of \( C \) that lead to good states from good states)

Based on the results obtained in (a), we want to show that \( T^W \) also simulates \( C \otimes G_W \) based on the fact that \( C \) is constructed from \( C^0 \) and that \( C \) has no bad states and no transitions that lead to bad states.

From Algorithm 3.1, \( C \) is built from \( C^0 \) by making bad states unreachable and removing all transitions that lead to bad states.
(i) Based on Lemma 3.5.1 and Lemma 3.5.2, we define another relation $R \upharpoonright S_C$ a projection of the states of $S_C$ into $R$ given by:

$$R \upharpoonright S_C = R \setminus \{(((s_n, t_n), s_n), t_n) \mid ((s_n, t_n), s_n) \notin S_{C \otimes G_W}\}$$

i.e., the set of pairs in $R$ excluding those not in $S_{C \otimes G_W}$. We also note that $(R \upharpoonright S_C) \subseteq R$.

(ii) Now, we are ready to show that $R \upharpoonright S_C$ is a simulation relation between $C \otimes G_W$ and $T^W$ if no transition in $C$ leads to a bad state.

Consider $(((s_n, t_n), s_n), t_n) \in R \upharpoonright S_C$ for some states $((s_n, t_n), s_n) \in S_{C \otimes G_W}$ and $t_n \in S_{T^W}$, and $(s_n, t_n) \in S_{C_{Good}}$, then from (i) we have that $(((s_n, t_n), s_n), t_n) \in R$. 
Now, suppose

\[
((s_n, t_n), s_n) \xrightarrow{\delta_n\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_C \otimes G_W \text{ and } (s_{n+1}, t_{n+1}) \in S_{C}^{Good}
\]

\[
\implies ((s_n, t_n), s_n) \xrightarrow{\delta_n\delta_n'\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C^0} \otimes G_W \text{ with } (g_n \leq g_n'), \text{ and }
\]

\[
(s_{n+1}, t_{n+1}) \in S_{C^0}^{Good} \text{ (by the construction of } C \text{ from } C^0)
\]

\[
\implies \exists t_{n+1} \in S_{\mathcal{T}W} \text{ such that } t_n \xrightarrow{\delta_n[g_n']} t_{n+1} \in \Gamma_{\mathcal{T}W} \text{ with } (g_n \leq g_n'),
\]

\[
\text{and } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \text{ (by the fact that } R \text{ is a simulation relation between } C^0 \otimes G_W \text{ and } \mathcal{T}W)\]

\[
\implies \exists t_{n+1} \in S_{\mathcal{T}W} \text{ such that } t_n \xrightarrow{\delta_n[g_n']} t_{n+1} \in \Gamma_{\mathcal{T}W} \text{ with } (g_n \leq g_n'),
\]

\[
\text{and } (((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \upharpoonright S_C \text{ (by definition of } R \upharpoonright S_C)\]

\[
\implies \exists t_{n+1} \in S_{\mathcal{T}W} \text{ such that } t_n \xrightarrow{\delta_n[g_n']} t_{n+1} \in \Gamma_{\mathcal{T}W} \text{ and }
\]

\[
(((s_{n+1}, t_{n+1}), s_{n+1}), t_{n+1}) \in R \upharpoonright S_C \text{ (by definition of simulation relation on } g_n \text{ and } g_n') \]

That is, \( R \upharpoonright S_C \) is a simulation relation between \( (C \setminus S_C^{Bad}) \otimes G_W \) and \( \mathcal{T}W \). So now we have that every transition that leads to a good state from a good state in \( C \otimes G_W \) can be simulated by \( \mathcal{T}W \). Hence, we only have left to show that \( C \) has no bad states and has no transitions that lead to bad states from a good state.

(c) (Transitions of \( C \) that lead to bad states from a good state)

Here, we show that all bad states and transitions that lead to bad states are eliminated from \( C \) during the construction of \( C \) from \( C^0 \). That is, \( C \) has no
bad states and has no transitions that lead to bad states. For example, the
synthesis of the controller $C$ in Figure 3.10(g) from $C^0$ in Figure 3.10(c) resulted
in the elimination of all transitions that lead to bad states and transitions that
violate controllability (Definition 3.3.8).

Now, consider

$$T = ((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_C \otimes \Gamma_W$$

(3.2)

and $(s_{n+1}, t_{n+1}) \in S_{C}^{Bad}$ such that $T$ is a transition in the sequence (execution)

$$s_0 \xrightarrow{\delta_0[g_0]} s_1 \xrightarrow{\delta_1[g_1]} s_2 \ldots s_{n-1} \xrightarrow{\delta_{n-1}[g_{n-1}]} s_n$$

In cases 1, 2, 3 and 4 below, we show that the final output of Algorithm 3.1
given by $C$ has no bad states and has no transitions that result in bad states.
That is, applying Algorithm 3.1 to $C^0$ results in the elimination of all bad states
and transitions that lead to bad states. We show that (2) is never the case.
There are three possible cases based on control decision and the kind of tran-
sition during the synthesis of $C$ from Algorithm 3.1. That is, given a transition
$T$ which leads to a bad state in $C$, then after applying Algorithm 3.1 $T$ would
be eliminated.

(I) Case 1 (Static controllability)

We assume that $T$ is a static transition which leads to a bad state in $C$ and
we consider the case where $T$ is uncontrollable and the case where $T$ is controllable. In the case that $T$ is an uncontrollable transition, it implies that $T$ is not enabled in $T^W$ due to the fact that $T^W$ is controllable. Now since $T$ is not enabled in $T^W$, it follows that Algorithm 3.1 would have eliminated $T$ during the construction of $C$. Hence it would be a contradiction if it holds that $T$ is in $C$. A similar situation occurs in the case that $T$ is a controllable transition and leads to a bad state. It follows that $T$ is not enabled in $T^W$, hence Algorithm 3.1 would have disabled $T$ to prevent it from occurring. Hence, it contradicts the hypothesis that $T$ is in $C$. We formally prove the above two cases in the following two steps, respectively.

- Suppose that $T$ is a static transition where $g_n = true$ and let $\delta_n \in \Sigma_{uc}$.

Then, from (2) we have

$$(s_n, t_n, s_n) \xrightarrow{\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma^c \otimes \mathcal{G}_W$$

and

$$(s_{n+1}, t_{n+1}) \in \mathcal{S}^{Bad}_C$$

from the definition of $\otimes$.

(since $T^W$ is controllable, any bad state and transition leading to a bad state which is not in $C$ is also not in $T^W$)

Then,

$$(s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma^c \text{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}^{Bad}_C$$

(assuming $T^W$ is controllable).

Then we get

$$(s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \mathcal{S}^{Bad}_C$$

(since $T^W$ is controllable, any bad state and transition leading to a bad state which is not in $C$ is also not in $T^W$)

$$(s_{n+1}, t_{n+1}) \notin \Gamma^c \text{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}^{Bad}_C$$

Then

$$(s_{n+1}, t_{n+1}) \in \mathcal{S}^{Bad}_C$$

(since $T^W$ is controllable, any bad state and transition leading to a bad state which is not in $C$ is also not in $T^W$)
(δₙ will not be enabled at tₙ of ℰₚ (which violates the definition of static controllability))

⇒ T would have been eliminated from C

by Algorithm 3.3 (Lines 4–20)

(hence, it will be a contradiction to say that (sₙ, tₙ) δₙ→ (sₙ₊₁, tₙ₊₁) ∈ Γₚ⊗ecidedₚ)

This step can be seen from the construction of C by Algorithm 3.3 (Lines 4–20), where any uncontrollable transition T enabled at sₙ in ℰₚ but not enabled at a corresponding state (sₙ, tₙ) of C will lead to the elimination of (sₙ, tₙ) from the states of C (Line 14). The state (sₙ, tₙ) will not even exist in the set of states of C.

• Similarly, suppose that T is a static transition where gₙ = true and
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

let $\delta_n \in \Sigma_c$, then from (2) we have

$$((s_n, t_n), s_n) \xrightarrow{\delta_n} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes G_w} \text{ and } (s_{n+1}, t_{n+1}) \in S^B_{C}$$

$$\implies (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_{C} \text{ and } (s_{n+1}, t_{n+1}) \in S^B_{C}$$

(from the definition of $\otimes$)

$$\implies t_n \xrightarrow{\delta_n} t_{n+1} \notin \Gamma_t$$

(any static controllable transition that leads to a bad state in $C$ is disabled in $T^W$)

$$\implies \delta_n \notin E^S_{T^W}(t_n)$$

($\delta_n$ will not be enabled at $t_n$ of $T^W$)

$$\implies T \text{ will be disabled by Algorithm 3.3 (Line 24)}$$

(since $\delta_n \in \Sigma_c$)

$$\implies \text{It will be a contradiction to say that } T \in \Gamma_{C \otimes G_w}$$

That is, $T$ will be disabled if it leads to a bad state. Hence, no controllable transition will end up in a bad state.

(II) Case 2 (Dynamic controllability 1, stronger guards generation)

Here we assume that $T$ is a dynamic type 1 transition which leads to a bad state in $C$ and we consider the case where $T$ is uncontrollable and the case where $T$ is controllable. In the case that $T$ is an uncontrollable transition, it implies that $T$ is either not allowed in $T^W$ at all or it allowed but the guards on both transitions are not satisfied as a result of the controllability of $T^W$. It follows that Algorithm 3.1 would have strengthened the guard on $T$ during the construction of $C$ and making the guard on
T stronger, which implies that the state \((s_{n+1}, t_{n+1})\) is not reachable. It follows that \(T\) is not a valid transition. Hence, it would be a contradiction if it holds that \(T\) is in \(C\). A similar scenario holds in the case that \(T\) is a controllable transition and leads to a bad state. It follows that \(T\) is not allowed in \(T^W\) at all or the guards on both transitions do not agree. Now, since \(T\) is controllable, Algorithm 3.1 would have disabled \(T\) when the guards are not satisfiable preventing it from occurring. Hence, it contradicts the hypothesis that \(T\) is in \(C\). We formally prove the above two cases in the following two steps, respectively.

- Suppose that \(T\) is a dynamic type 1 transition and \(\delta_n \in \Sigma_{uc}\) (e.g., in Example 3.5.1 take \(T\) to be the transition \(((s_2, t_2) \xrightarrow{\text{atom}_1(x, y)[3 \leq x < 7]} \text{BAD}_1)\) of Figure 3.10(c)).
Then, from (2) we have,

\[(s_n, t_n, s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes G_W} \text{ and } (s_{n+1}, t_{n+1}) \in S_C^{Bad}\]

\[\implies (s_n, t_n, s_n) \xrightarrow{\delta_n[g_n]} (s_{n+1}, t_{n+1}) \in \Gamma_C \text{ and } (s_{n+1}, t_{n+1}) \in S_C^{Bad}\]

(from the definition of $\otimes$)

\[\implies \exists (t_n \xrightarrow{\delta_n[g'_n]} t_{n+1}) \in \Gamma_{TW} \text{ such that } (g'_n \land g_n) \text{ is satisfiable}\]

(since $\mathcal{T}^W$ is controllable and the from the contruction of $C$)

\[\implies (\delta_n, g_n) \notin E_{TW}^p(t_n)\]

($T$ would not be enabled at the state $t_n$ of the $\mathcal{T}^W$)

\[\implies \text{the guard } g_n \text{ of } C \text{ would have been strengthened and transition } T \text{ eliminated}\]

By Algorithm 3.5 (Lines 38–46), the guards of $C$ are strengthened to that of $\mathcal{T}^W$, in particular the guard $g_n$ on $T = ((s_n, t_n), s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes G_W}$ will never be true and the state $(s_{n+1}, t_{n+1})$ will be eliminated. Hence, it will be a contradiction to say that $T \in \Gamma_{C \otimes G_W}$.

- Similarly, suppose that $T$ is a dynamic transition of type 1 and let
δ_n ∈ Σ_c, then from (2) we have

\[(s_n, t_n, s_n) \xrightarrow{\delta_n[g_n]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{c \otimes g_w} \text{ and } (s_{n+1}, t_{n+1}) \in S_C^{Bad}\]

\[\Rightarrow (s_n, t_n) \xrightarrow{\delta_n[g_n]} (s_{n+1}, t_{n+1}) \in \Gamma_C \text{ and } (s_{n+1}, t_{n+1}) \in S_C^{Bad}\]

(from the definition of \(\otimes\))

\[\Rightarrow \#(t_n \xrightarrow{\delta_n[g'_n]} t_{n+1}) \in \Gamma_{TW} \text{ such that } (g'_n \land g_n) \text{ is satisfiable}\]

(i.e., a controllable transition that leads to a bad state in C means either the transition is in TW but the guards are not satisfied for all values or the transition is not in TW at all)

\[\Rightarrow T \text{ will be disabled by Algorithm 3.6 (Line 6)}\]

(since δ_n ∈ Σ_c)

\[\Rightarrow \text{It will be a contradiction to say that } T \in \Gamma_{c \otimes g_w}\]

(III) Case 3 (Dynamic controllability type 2, event enforcement)

Suppose that T is a dynamic type 2 transition which leads to a bad state in C. In this particular case (Case 3) the guard on T has a variable whose value depends on the output of an atomic operation (e.g., in Figure 3.6).

We consider separately the case where T is uncontrollable and the case where T is controllable.

In the case that T is an uncontrollable transition, it implies that T is
either not allowed in $\mathcal{T}^W$ at all or it allowed but the guards on both transitions are not satisfied as a result of controllability of $\mathcal{T}^W$. Now, by controllability of event enforcement, there must exist another enforceable transition, say $T'$, also enabled at the same state as $T$ to be used to pre-empt $T$ during runtime. This means that $(s_{n+1}, t_{n+1})$ is unreachable. It follows that $T$ is not a valid transition in $\mathcal{C}$. This contradicts the hypothesis that $T$ is in $\mathcal{C}$.

Similarly, consider the case that $T$ is a controllable transition and leads to a bad state. It follows that $T$ is not allowed in $\mathcal{T}^W$ at all or the guards on both transitions do not agree. Now, since $T$ is a controllable transition, Algorithm 3.1 would have disabled $T$ when the guards were not satisfiable preventing it from occurring. Hence, it contradicts the hypothesis that $T$ is in $\mathcal{C}$. We formally prove the above two cases in the following two steps, respectively.

- Recall that $g_n^A(v)$ denote a guard $g_n$ with a variable $v$ whose values depend on an output of an atomic operation $A$ of a given transition system. Suppose that $T$ is a dynamic type 2 transition where $g_n = g_n^A(v)$ and let $\delta_n \in \Sigma_{uc}$ (e.g., in Example 3.5.1 take $T$ to be the transitions $(s_4, t_4) \xrightarrow{\text{atom}_2(x, y)[y=a]} (BAD_2)$ of Figure 3.10(c)). Then, from (2) we
have,

\[ ((s_n, t_n), s_n) \xrightarrow{\delta_n[g^A_n(v)]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes G^W} \text{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}^B_{C} \]

\[ \implies (s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1}) \in \Gamma_C \text{ and } (s_{n+1}, t_{n+1}) \in \mathcal{S}^B_{C} \]

(from the definition of \( \otimes \))

\[ \implies \exists (t_n \xrightarrow{\delta_n[g^A_n(v)_n]} t_{n+1}) \in \Gamma_{T^W} \text{ such that } ((g^A_n(v))' \land g^A_n(v)) \]

is satisfiable

(since \( T^W \) is controllable and from the construction of \( C \))

\[ \implies (\delta_n, g^A_n(v)) \notin E^D_{T^W}(t_n) \]

\( (T \) is not enabled at the state \( t_n \) of \( T^W \))

\[ \implies \exists \delta'_n : \delta'_n \in E^D_{T^W}(t_n) \text{ and } \delta'_n \in \Sigma_f \]

by Corollary 3.3.1, i.e., from the controllability of dynamic transition of type 2 using event enforcement. Otherwise Algorithm 3.5 (Lines 56–57) would have eliminated \((s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1})\) from \( C \). Since there is an enforceable event \( \delta'_n \) also enabled at state \((s_n, t_n)\) of the controller, the state \((s_{n+1}, t_{n+1})\) is not reachable because \( \delta'_n \) would be used to preempt \( T \) at runtime.

\[ \implies \text{It will be a contradiction to say that } T \in \Gamma_{C \otimes G^W} \]
This can be seen in Lines 49–61 of Algorithm 3.5, if there is a state \((s_n, t_n)\) in the set of states of \(\mathcal{C}\) with a transition \(T\) whose guard is given by \(g_n^A(v)\) and leads to a bad state, it implies that there must be an enforceable transition \(T'\) exiting \((s_n, t_n)\), otherwise the state \((s_n, t_n)\) is removed from the states of \(\mathcal{C}\). Hence, it will be a contradiction to say that \(T \in \Gamma_{C \otimes G_W}\). Algorithm 3.5 monitors the variable \(v\) at runtime and ensures that \(\delta_n'\) is enforced.

- Similarly, suppose that \(T\) is a dynamic transition of type 2 where \(g_n = g_n^A(v)\) and let \(\delta_n \in \Sigma_c\), then from (2) we have

\[
\begin{align*}
((s_n, t_n), s_n) & \xrightarrow{\delta_n[g_n^A(v)]} ((s_{n+1}, t_{n+1}), s_{n+1}) \quad \text{in } \Gamma_{C \otimes G_W} \quad \text{and} \quad (s_{n+1}, t_{n+1}) \in S_{Bad}^C \\
\implies (s_n, t_n) & \xrightarrow{\delta_n[g_n^A(v)]} (s_{n+1}, t_{n+1}) \in \Gamma_C \quad \text{and} \quad (s_{n+1}, t_{n+1}) \in S_{Bad}^C \\
\quad \text{(from the definition of } \otimes) \\
\implies \nexists (t_n \xrightarrow{\delta_n[(g_n^A(v))']} t_{n+1}) \in \Gamma_{T_W} \quad \text{such that} \\
((g_n^A(v))' \land g_n^A(v)) \quad \text{is satisfiable}
\end{align*}
\]
(a dynamic type 2 controllable transition that leads to a bad state in \( C \) means either the transition is in \( T^W \) but the guards are not satisfied for all values or the transition is not in \( T^W \) at all)

\[ \Longrightarrow T \] will be disabled by Algorithm 3.6 (Line 6)

(since \( \delta_n \in \Sigma_c \))

\[ \Longrightarrow \] It will be a contradiction to say that \( T \in \Gamma_{C \otimes G^W} \)

(IV) Finally, in the computation of \( C \) by Algorithm 3.1 (Line14), any new transition that was created as a result of the checking for controllability is removed.

From cases I, II, III, IV, we have shown that any transition \( T \in \Gamma_{C^0} \) that leads to a bad state is eliminated from the set of transitions of \( C^0 \) upon termination of Algorithm 3.1. Thus, the final output of Algorithm 3.1 given by \( C \) has no bad states and has no transitions that lead to a bad state. This implies that every transition in the controlled system \( C \otimes G^W \) leads to a good state which can be matched by \( T^W \). Hence, \( C \otimes G^W \leq T^W \).

In the following we prove the reverse case of the theorem. Part two of the proof of this theorem is to show that given that \( T^W \leq G_W \) and if the specification simulates the transition system of the plant coupled with the controller, then it implies that \( G_W \) is controllable with respect to the \( T^W \). This is formally stated in the box below.
Only if: Assume $T^W \preceq G_W$ and $C \otimes G_W \preceq T^W$.

To prove:

$T^W$ is controllable.

To prove the claim we shall consider two cases based on the type of transitions and the given assumption $C \otimes G_W \preceq T^W$ to prove controllability. We note that no matter what algorithm is used to construct $C$ the theorem must be satisfied.

- **Static Case:**

  Here we prove the first part of controllability (i.e., controllability of static transitions) by using the fact that if there is a simulation relation between $C \otimes G_W$ and $T^W$, then it implies every static transition that is enabled in $C \otimes G_W$ is also enabled in $T^W$, which proves static controllability. We show
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

Given $C \otimes G_W \leq \mathcal{T}^W \implies$

[there exists a relation $R \subseteq S_{(C \otimes G_W)} \times S_{\mathcal{T}^W}$ such that $\forall ((p, r), q) \in R$, if $(p, r) \delta \rightarrow (p', r') \in \Gamma_{C \otimes G_W} \implies \exists q'$ such that $q \delta \rightarrow q' \in \Gamma_{\mathcal{T}^W}$ and $((p', r'), q') \in R$]

(from the definition of simulation relation)

$$\implies \forall \delta \in \Sigma, ((p, r), q) \in R \subseteq S_{(C \otimes G_W)} \times S_{\mathcal{T}^W} :$$

$$(p, r) \delta \rightarrow (p', r') \in \Gamma_{C \otimes G_W} \implies \exists q'$ such that $q \delta \rightarrow q' \in \Gamma_{\mathcal{T}^W}$$

and $((p', r'), q') \in R \subseteq S_{(C \otimes G_W)} \times S_{\mathcal{T}^W}$$

$$\implies \forall \delta \in \Sigma_{uc} : (p, r) \delta \rightarrow (p', r') \in \Gamma_{C \otimes G_W} \implies \exists q'$ such that $q \delta \rightarrow q' \in \Gamma_{\mathcal{T}^W}$]

(since $\Sigma_{uc} \subseteq \Sigma$)

$$\implies \forall \delta \in \Sigma_{uc} : \delta \in E_{C \otimes G_W}(p, r) \implies \delta \in E_{\mathcal{T}^W}(q)$$

- Dynamic Case:

Similarly, we prove the second part of controllability (i.e., dynamic controllability) by using the fact that if there is a simulation relation between $C \otimes G_W$ and $\mathcal{T}^W$, then it implies every dynamic transition that is enabled in $C \otimes G_W$ is also enabled in $\mathcal{T}^W$ and the guards are satisfied. In this case, no matter how $C$ was constructed and whether the construction uses event enforcement techniques or not, it must ensure that the guards on both transitions of $C \otimes G_W$ and $\mathcal{T}^W$ are satisfiable. We show this formally as follows by considering both
dynamic 1 and dynamic 2 transitions together.

Given $C \otimes G \preceq T^W$, there is a relation $R \subseteq S_{(C \otimes G)W} \times S_{T^W}$ such that $\forall ((p, r), q) \in R,$ if $(p, r) \xrightarrow{\delta_{[g_1]}} (p', r') \in \Gamma_{C \otimes G} \Rightarrow \exists q' such that $q \xrightarrow{\delta_{[g_2]}} q' \in \Gamma_{T^W}$, where $g_1 \leq g_2$

and $((p', r'), q') \in R$

(from the definition of simulation relation)

$\Rightarrow [\forall \delta \in \Sigma, ((p, r), q) \in R \subseteq S_{(C \otimes G)W} \times S_{T^W} :$

$(p, r) \xrightarrow{\delta_{[g_1]}} (p', r') \in \Gamma_{C \otimes G} \Rightarrow \exists q' such that $q \xrightarrow{\delta_{[g_2]}} q' \in \Gamma_{T^W}$, where $g_1 \leq g_2$

and $((p, r), q) \in R \subseteq S_{(C \otimes G)W} \times S_{T^W}]$

$\Rightarrow [\forall \delta \in \Sigma_{uc} : (p, r) \xrightarrow{\delta_{[g_1]}} (p', r') \in \Gamma_{C \otimes G} \Rightarrow \exists q' such that $q \xrightarrow{\delta_{[g_2]}} q' \in \Gamma_{T^W}$ where $g_1 \leq g_2]$

(since $\Sigma_{uc} \subseteq \Sigma$)

$\Rightarrow [\forall \delta \in \Sigma_{uc} : (\delta, g_1) \in E_{C \otimes G}^P(p, r) \Rightarrow [(\delta, g_2) \in E_{T^W}^P(q) such that \ (g_1 \leq g_2)]$

From the above two steps it implies that controllability is satisfied, which completes the proof of the theorem.

3.5.2 Minimally Restrictive Controller

Before we introduce and prove the next theorem let consider the following lemmata.

Lemma 3.5.4. Given a plant $G^W$ and a specification $T^W$, the controller $C$ generated by Algorithm 3.1 will satisfy $C \preceq G^W$. 
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

Proof. The proof follows from the fact that the specification $T^W$ is simulated by the plant, i.e., $T^W \preceq G_W$ and since $C \preceq T^W$, it implies that $C \preceq G_W$. ■

Lemma 3.5.5. Let $C^0 = G_W \times_{ref} T^W$, the controller $C$ generated by Algorithm 3.1 will satisfy $C \preceq C^0$.

Proof. To prove this lemma, we show that $C$ is constructed from $C^0$ and that any new transition or state added is removed before the termination of Algorithm 3.1.

First, Line 6 of Algorithm 3.1 computes $C^0 = G_W \times_{ref} T^W$. Let $C^k$ denote some $k$ ($k \geq 1$) iterations of the repeat loop over $C^0$. Next, the repeat until loop from Line 8 to Line 13 iterates over the $C^{k-1}$ and removes some states and transitions or adds new bad states and transitions that lead to these bad states. In the case that states and transitions are removed from $C^{k-1}$, then $C$ will still be simulated by $C^0$. On the other hand, if new bad states and transitions are added to from $C^{k-1}$ then Algorithm 3.1 removes these bad states and transitions at Line 14 and from this point until the termination of algorithm no new states or transitions are added to $C^k$. Thus, $C$, which is the output of Algorithm 3.1, is simulated by $C^0$, which completes the proof. ■

Lemma 3.5.6 (Transitions that lead to bad state in $C^0$). Let $C^0 = G_W \times_{ref} T^W$ and let $(s, t) \xrightarrow{\delta | g} (s', t') \in \Gamma_{C^0}$ where $(s', t') \in S^\text{Bad}_{C^0}$, then either $\exists g' \in B : (\delta, g') \in E^D_{G_W}(s)$ but $(\delta, g) \notin E^D_{C^0 \setminus S^\text{Bad}_{C^0}}(s', t')$ and $(g' \land g)$ satisfiable. That is, if a transition leads to a bad state in $C^0$, then it implies that this transition is enabled at the plant state $s$ but this transition is not enabled at the corresponding state $(s', t')$ in $C^0 \setminus S^\text{Bad}_{C^0}$. 
Proof. The proof of Lemma 3.5.6 is a direct consequence of Definition 3.3.5.

Note that in Lemma 3.5.6, if $\delta$ is uncontrollable then it means that $(s,t) \xrightarrow{\delta|g} (s',t')$ violates the definition of dynamic controllability.

Now, we state the theorem of minimally restrictive controller. Let $A \prec B$ denote a strict simulation relation between $A$ and $B$ such that $B$ simulates $A$, but $A$ does not simulate $B$.

**Theorem 3.5.2 (Minimally Restrictive Controller).** Given an SLTS $G_W = (S_G, S_0_G, I_G, O_G, A_G, \Gamma_G, S^F_G, V_G, B_G)$ and a set of specifications $T_W$, the controller $C$ generated by Algorithm 3.1 is a minimally restrictive nonblocking and communication-error free controller for $G_W$ satisfying $C \otimes G_W \preceq T_W$.

The proof of Theorem 3.5.2 requires proving the following claim:

Given that $C = Composer(G_W, T_W)$ is the output of Algorithm 3.1, then

$$\nexists C' \text{ such that } C \otimes G_W \prec C' \otimes G_W$$

where $C'$ is assumed to be a controller which satisfies all controllable, nonblocking and communication-error free conditions and $C' \otimes G_W \preceq T_W$.

Proof. We proceed by contradiction.

(i) Suppose that $C$ is not minimally restrictive controller. This implies the following:

$$\exists C' : C \otimes G_W \prec C' \otimes G_W$$ (3.3)
where $C'$ is assumed to be a controller which satisfies all controllable, non-blocking and communication-error free conditions and $C' \otimes G_W \leq T^W$.

(ii) By (3.3), there exists a simulation relation $R$ between $C \otimes G_W$ and $C' \otimes G_W$ such that

\[
(C \otimes G_W) \prec (C' \otimes G_W) \quad (3.4)
\]

(iii) By (3.4), it follows that there exists an execution

\[
e' = (c'_0, s_0) \xrightarrow{\delta_0[g_0]} (c'_1, s_1) \xrightarrow{\delta_1[g_1]} (c'_2, s_2) \rightarrow \ldots \rightarrow (c'_n, s_n) \xrightarrow{\delta_n[g_n]} (c'_{n+1}, s_{n+1}) \quad (3.5)
\]

in $C' \otimes G_W$ and there exists a matching execution

\[
e = (((s_0, t_0), s_0) \xrightarrow{\delta_0[g_0]} ((s_1, t_1), s_1) \xrightarrow{\delta_1[g_1]} ((s_2, t_2), s_2) \rightarrow \ldots \rightarrow ((s_{n-1}, t_{n-1}), s_{n-1}) \xrightarrow{\delta_{n-1}[g_{n-1}]} ((s_n, t_n), s_n) \quad (3.6)
\]

(\text{where } (((s_i, t_i), s_i), (c'_i, s_i)) \in R \text{ and } ((s_i, t_i), s_i) \in S_{C' \otimes G_W}, (c'_i, s_i) \in S_{C' \otimes G_W}, s_i \in S_{G_W}, t_i \in S_{T^W}, c'_i \in S_{C'} \text{ for all } 0 \leq i \leq n)\]

that is also valid in $C \otimes G_W$ but there is no $t_{n+1} \in S_{T^W}$ and there is no $g_m \in B$ such that

\[
(((s_n, t_n), s_n) \xrightarrow{\delta_n[g_m]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C' \otimes G_W} \text{ and } (g_n \land g_m) \text{ satisfiable})
\]

We will show that (iii) leads to a contradiction.
(iv) The proof strategy we employ here is to consider the last transition

\[(c'_n, s_n) \xrightarrow{\delta_n[g_n]} (c'_{n+1}, s_{n+1}) \in \Gamma_{C' \otimes \mathcal{G}_W} \text{ from (3.5)}.\]

Then, we show that if there is no \(t_{n+1}\) and there is no \(g_m\) such that

\[((s_n, t_n), s_n) \xrightarrow{\delta_n[g_m]} ((s_{n+1}, t_{n+1}), s_{n+1}) \in \Gamma_{C \otimes \mathcal{G}_W} \text{ and } (g_n \land g_m) \text{ satisfiable}. \tag{3.7}\]

Then, \((c'_n, s_n) \xrightarrow{\delta_n[g_n]} (c'_{n+1}, s_{n+1}) \notin \Gamma_{C' \otimes \mathcal{G}_W}\]

which contradicts Equation (3.5)

Now, to show (3.7) requires showing that

\[\not\exists t_{n+1}, \not\exists g_m \text{ such that } (s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1}) \in \Gamma_{C} \tag{3.8}\]

and \((g_n \land g_m) \text{ satisfiable}.\]

(by definition of \(\otimes\) and Lemma 3.5.4)

We prove (3.8) by showing that in the construction of \(C\) by Algorithm 3.1 there is no \(t_{n+1}\) and there is no \(g_m\) such that \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) is in \(C\) as a result of the following reasons.

Now, in Algorithm 3.1 (Line 6), \(C\) is initially constructed from \(C^0 = \mathcal{G}_W \times_{\text{ref}} \mathcal{T}_W\) and then various reduction techniques are applied to \(C^0\) to derive \(C\). Therefore, we run Algorithm 3.1 on \(\mathcal{G}_W\) and \(\mathcal{T}_W\), by taking note of how the algorithm removes \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) or why \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) does not exist in \(C\).

There are two main cases to consider, the case where \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1}) \notin C\)
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

Γ₀ and the case where \((s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \in Γ₀\).

1. Case I: Composition Refinement \( ((s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \notin Γ₀ \) by Algorithm 3.1 Line 6)

Now, suppose that in the initial construction of \( C \) from \( C₀ \) by Algorithm 3.1 Line 6

\[ \not\exists t_{n+1}, \not\exists g_m \text{ such that } (s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \in Γ₀ \text{ and } (g_n \land g_m) \text{ satisfiable.} \]

Then, it implies that \( (s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \notin Γ_C \).

2. Case II: \( ((s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \in Γ₀ \) by Algorithm 3.1 Lines 7–19

Next, we consider the case where in the initial construction of \( C₀ \) we have \( (s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \in Γ₀ \) and \( (g_n \land g_m) \) satisfiable and let \( C_k \) denote the resultant SLTS obtained after some \( k \) iterations of the repeat until loop of Algorithm 3.1 (Lines 8–13) on \( C₀ \), where \( 1 \leq k \leq m \) and \( m \) is the number of iterations of the repeat until loop upon termination. In the following sub-cases, we run the rest of Algorithm 3.1 Lines 7–19 on \( C^{k-1} \) noting where \( (s_n, t_n) \xrightarrow{δ_n[g_m]} (s_{n+1}, t_{n+1}) \) is eliminated from \( C^{k-1} \).

Stages 1-4 below deal with elimination of a transition done within the repeat until loop of Algorithm 3.1 (Lines 8–13), whereas Stages 5-6 deal with elimination of a transition after the termination of the repeat until loop (Algorithm 3.1 (Lines 14–19)).
(a) **Stage 1: Elimination of Static Transitions (Algorithm 3.3)**

Suppose that \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) is a static transition where \(g_m = \text{true}\) (i.e., \((s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1})\)) then, in this stage of the algorithm \((s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1})\) may be eliminated from the set of transitions of \(C^{k-1}\) due to the following:

Algorithm 3.3 iterates over each state \(p\) of \(G_W\) and a corresponding state \((p, q)\) of \(C^{k-1}\) and checks that if there exists a static transition \(t\) that is enabled at state \(p\) of \(G_W\) but not enabled in the corresponding state \((p, q)\) of \(C^{k-1}\) and then, if \(t\) is an uncontrollable transition, Lines 14–17 of Algorithm 3.3 eliminate \(t\) and if \(t\) is a controllable transition, Line 24 of Algorithm 3.3 marks \(t\) as disabled.

In particular, given \((s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1})\) the transition under consideration, the for loop of Algorithm 3.3 Lines 4–5 check if

\[
\exists \alpha \in \Sigma \text{ such that } \alpha \in E^S_{G_W}(s_{n+1}) \text{ but } \\
\n\n\exists \alpha \text{ such that } \alpha \in E^S_{C^{k-1}}(s_{n+1}, t_{n+1})
\]

Then, one of the following cases holds:
3.5. PROOF OF CORRECTNESS AND COMPLETENESS 180

- **Case 1:** \( \delta \) is uncontrollable, i.e., \( \alpha \in \Sigma_{uc} \)

\[ \implies \exists \alpha \in \Sigma_{uc} \text{ such that } \alpha \in E_{gw}^S (s_{n+1}) \text{ but } \not\exists \alpha \text{ such that } \alpha \in E_{c_k-1}^S(s_{n+1}, t_{n+1}) \]

(from (3.9))

\[ \implies \text{Algorithm 3.3 Lines 14–17 will eliminate the state } (s_{n+1}, t_{n+1}) \]

from \( C^{k-1} \) and all associated transitions

(since the previous case violates the definition of static controllability)

\[ \implies \exists (t_{n+1}, s_{n+1}) \text{ such that } (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_{c_k} \]

(since \( (s_{n+1}, t_{n+1}) \) has been eliminated from \( C^{k-1} \))

\[ \implies \exists (t_{n+1}, s_{n+1}) \text{ such that } (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \in \Gamma_c \]

(since \( (s_n, t_n) \xrightarrow{\delta_n} (s_{n+1}, t_{n+1}) \) has does not exists in \( C^k \))

- **Case 2:** \( \delta \) is controllable, i.e., \( \alpha \in \Sigma_c \)

\[ \implies \exists \alpha \in \Sigma_c \text{ such that } \alpha \in E_{gw}^S (s_{n+1}) \text{ but } \not\exists \alpha \text{ such that } \alpha \in E_{c_k-1}^S(s_{n+1}, t_{n+1}) \]

(from (3.9))

\[ \implies \text{Algorithm 3.3 Line 24 will disable } \alpha \text{ at state } (s_{n+1}, t_{n+1}) \text{ in } C^{k-1} \]
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

This means that when $C$ is combined with the plant $G_W$, $\alpha$ remains disabled.

(b) **Stage 2: Elimination of Blocking Transitions (Algorithm 3.4)**

This stage of the algorithm will check for blocking in $C^{k-1}$ and eliminate any transition that results in blocking.

In particular, Algorithm 3.4 Line 9 will check whether $(s_{n+1}, t_{n+1})$ is a final state or there is a sequence of transitions from $(s_{n+1}, t_{n+1})$ that leads to a final state.

In the case that this condition is not met, then Algorithm 3.4 Lines 23–28 eliminates $(s_{n+1}, t_{n+1})$ and all associated transitions from $C^{k-1}$, which implies that $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$ will not be in final output of Algorithm 3.1 $C$ when the algorithm terminates.

(c) **Stage 3: Elimination of Dynamic Type 1 Transitions (Algorithm 3.5 Lines 38–46 and 62)**

Now, we consider the case where $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$ is a dynamic type 1 transition. Then, at this stage of the algorithm $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$ is eliminated from the set of transitions of $C^{k-1}$ due to the following.

According to Algorithm 3.5, the set of states of $C^{k-1}$ is partitioned into
the set of good states $S_{C_{k-1}}^{Good}$ and the set of bad states $S_{C_{k-1}}^{Bad}$. Generally, Algorithm 3.5 (the for each loops at Lines 38–39) checks if a dynamic type 1 transition $t$ leads into a bad state. Then, in the case that $t$ is uncontrollable, by Lemma 3.5.6 it implies that $t$ violates the definition of dynamic controllability. Thus, Lines 40–42 in Algorithm 3.5 strengthen the guard to prevent the bad state from being reached. On the other hand, if $t$ is controllable, by Lemma 3.5.6 it implies a controllable transition is allowed in the plant but not in $C_{k-1}$, but because $t$ is controllable Algorithm 3.5 Line 62 will disable it. In particular given the transition $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$, Algorithm 3.5 (the for each loops at Lines 38–39) will first check whether $(s_{n+1}, t_{n+1})$ is a bad state or not: 

If $(s_{n+1}, t_{n+1}) \in S_{C_{k-1}}^{Bad}$, then one of the following cases holds:
3.5. PROOF OF CORRECTNESS AND COMPLETENESS

- **Case 1:** $\delta_n$ is uncontrollable, i.e., $\delta_n \in \Sigma_{uc}$

  \[ \implies \exists \delta_n \in \Sigma_{uc}, \exists g'_m : (\delta_n, g'_m) \in E_{G_{w}}^D(s_n) \text{ but } \not\exists \delta_n, \not\exists g_m : (\delta_n, g_m) \in E_{(C_{k-1} \setminus S_{k-1}^{bad})}^D(s_n, t_n) \text{ and } g'_m \land g_m \text{ satisfiable} \]

  (by Lemma 3.5.6)

  \[ \implies \text{Algorithm 3.5 Line 42 strengthens the guard } g_m \text{ so that it is never satisfiable since } (s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1}) \text{ does not satisfy dynamic controllability} \]

  \[ \implies (s_{n+1}, t_{n+1}) \text{ is unreachable in } C^k \]

  (since the guard $g_m$ was strengthened in $C^{k-1}$)

  \[ \implies (s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1}) \text{ will not be reachable in the final output } C \]

  of the algorithm

  (note: Stage 5 below eliminates these kind of states and associated transitions before the algorithm terminates)
Case 2: $\delta_n$ is controllable, i.e., $\delta_n \in \Sigma_c$

$$\implies \exists \delta_n \in \Sigma_c, \exists g'_m : (\delta_n, g'_m) \in E^D_{G_W} (s_n)$$

but $\nexists \delta_n, \nexists g_m : (\delta_n, g_m) \in E^D_{(C_{k-1}) \setminus S_{Bad}^{k-1}} (s_n, t_n)$ and $g'_m \land g_m$ satisfiable

(by Lemma 3.5.6)

$$\implies$$ Algorithm 3.5 Line 62 will mark $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$

as disabled in $C^k$

(since $\delta_n$ is controllable)

$$\implies (s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$$

is disable in the final output $C$ of the algorithm

(d) Stage 4: Elimination of Dynamic Type 2 Transitions (Algorithm 3.5 Lines 49–61 and 62)

Let $g^A_m(v)$ denote a guard $g_m$ with a variable $v$ whose values depend on an output of an atomic operation $A$ of a given transition system (e.g., in Figure 3.6, $A = func_1(x, y), v = y, g_m = (y < 10)$).

Here we consider the case where $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1}) \in \Gamma_{C_{k-1}}$ is a dynamic type 2 transition where $g_m = g^A_m(v)$, (i.e., $(s_n, t_n) \xrightarrow{\delta_n[g^A_m(v)]} (s_{n+1}, t_{n+1})$) then the algorithm will eliminate $(s_n, t_n) \xrightarrow{\delta_n[g^A_m(v)]} (s_{n+1}, t_{n+1})$

from $C^{k-1}$ due to the following:

Similar to Stage 3 of the algorithm, here Algorithm 3.5 (the two loops at Lines 49–50) will check whether the dynamic type 2 transition $(s_n, t_n) \xrightarrow{\delta_n[g^A_m(v)]} (s_{n+1}, t_{n+1})$ leads to a bad state, i.e., $(t_{n+1}, t_{n+1}) \in S_{Bad}^{C_{k-1}}$. In the case that
it leads to a bad state and $\delta \in \Sigma_{uc}$, then by Lemma 3.5.6, it implies that $(s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1})$ violates then definition of controllability. Thus, for this kind of transition Lines 51–53 in Algorithm 3.5 ensure that there is an enforcible event say $\delta' \in \Sigma_f$ also enabled at state $(s_n, t_n)$ to be used at runtime to preempt $(s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1})$ by monitoring the value of the variable $v$, otherwise the state $(s_n, t_n)$ will be eliminated by Algorithm 3.5 Lines 49–61 (Corollary 3.3.1). On the other hand, if $\delta \in \Sigma_c$, then by Lemma 3.5.6, we know that a dynamic 2 controllable transition is enabled in the plant but not in $\mathcal{C}^{k-1}$, however since $(s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1})$ is a controllable transition, Algorithm 3.5 Lines 62 will mark it as disabled.

More precisely, given $(s_n, t_n) \xrightarrow{\delta_n[g^A_n(v)]} (s_{n+1}, t_{n+1})$ as the transition under consideration, Algorithm 3.5 will first check whether $(s_{n+1}, t_{n+1})$ is a bad state or not:

If $(s_{n+1}, t_{n+1}) \in S_{\mathcal{C}^{k-1}}^{Bad}$ then one of the following cases holds,
• **Case 1:** $\delta_n$ is uncontrollable, i.e., $\delta_n \in \Sigma_{uc}$

$$\exists \delta_n \in \Sigma_{uc}, \exists (g_m^A(v))^\prime : (\delta_n, (g_m^A(v))^\prime) \in E_{G_W}^D(s_n) \text{ but}$$

$$\nexists \delta_n, \exists g_m^A(v) : (\delta_n, g_m^A(v)) \in E_{(C_{k-1}\setminus S_{k-1}(s_n, t_n)}^D \text{ and } ((g_m^A(v))^\prime \land g_m^A(v)) \text{ satisfiable}$$

(by Lemma 3.5.6)

$$\implies \text{Algorithm 3.5 (Lines 49–61) will ensure that } \exists \delta' \in \Sigma_f, \exists g_l$$

such that $(\delta', g_l) \in E_{(C_{k-1}\setminus S_{k-1}(s_n, t_n)}^D)$, otherwise the state $(s_n, t_n)$ would have been eliminated at Line 57 of Algorithm 3.5 (since the $(s_n, t_n) \xrightarrow{\delta_n[g_m^A(v)]} (s_{n+1}, t_{n+1})$ violates the definition of controllability and by Corollary 3.3.1)

$$\implies \delta' \text{ will be used to preempt } (s_n, t_n) \xrightarrow{\delta_n[g_m^A(v)]} (s_{n+1}, t_{n+1}) \text{ in } C^k$$

$$\implies \text{the state } ((s_{n+1}, t_{n+1})) \text{ is unreachable at runtime}$$

(since $\delta'$ is enforced at Line 53 of Algorithm 3.5)

$$\implies \text{state } (s_{n+1}, t_{n+1}) \text{ would not be reachable in the final output } C \text{ when the algorithm terminates (note: Stage 5 below eliminates these kind of states and associated transitions before the algorithm terminates)}$$
**Case 2:** $\delta_n$ is controllable, i.e., $\delta_n \in \Sigma_c$

\[
\implies \exists \delta_n \in \Sigma_c, \exists (g^A_m(v))': (\delta_n, (g^A_m(v))') \in E_{g^A_m}^D(s_n) \text{ but }
\not\exists \delta_n, \not\exists g_m : (\delta_n, g^A_m(v)) \in E_{(c_{k-1}\setminus S^{Bad}_{c_{k-1}})}^D(s_n, t_n)
\]

and $(g^A_m(v))' \land g^A_m(v)$ satisfiable
(by Lemma 3.5.6)

\[
\implies \text{Algorithm 3.5 Line 62 will disable } (s_n, t_n) \xrightarrow{\delta_n[g^A_m(v)]} (s_{n+1}, t_{n+1})
\]

(since $\delta_n$ is controllable)

\[
\implies (s_n, t_n) \xrightarrow{\delta_n[g^A_m(v)]} (s_{n+1}, t_{n+1})
\]

is marked as disabled in the final output $C$ of the algorithm.

Stages 5 and 6 below deal with the part of the algorithm after the termination of the repeat until loop of Algorithm 3.1. That is, in the following we run the rest of Algorithm 3.1 Lines 14–19 on $C^k$.

(e) **Stage 5: Elimination of Unreachable Bad States and all Associated Transitions (Algorithm 3.7)**

Once the repeat until loop of Algorithm 3.1 has terminated, all states in $S^{Bad}_{C^k}$ would have been made unreachable, thus Algorithm 3.7 removes every states in $S^{Bad}_{C^k}$ and their associated transition from $C^k$.

In particular, given $(s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})$ as the transition that we are considering, if $(s_{n+1}, t_{n+1})$ is a member of $S^{Bad}_{C^k}$ and has not already
been eliminated after the termination of the `repeat until` loop in Algorithm 3.1, then this stage of the algorithm (Algorithm 3.7 Lines 2–8) removes \((s_{n+1}, t_{n+1})\) and all its associated transitions. This is because, when the iteration of the `repeat until` loop terminates, all the states in the set of bad states \(S^{Bad}_{C^k}\) would have been made unreachable either by strengthening the guards as in (c) or by the use of event enforcement as in (d) or by disabling of events. This means, at this point the set of bad states \(S^{Bad}_{C^k}\) are not relevant in \(C^k\) and are removed by the algorithm.

(f) **Stage 6: Elimination of communication error transitions (Algorithm 3.2)**

Finally, Algorithm 3.2 eliminates all transitions that lead to communication errors. In particular, it will check whether \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) satisfies the constraints on unspecified receptions (Algorithm 3.2 Lines 22-26) or non-executable interactions (Algorithm 3.2 Lines 35-40) or not. In the case that it violates any of these constraints, then Algorithm 3.2 Line 27 and Line 41 eliminate \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) from \(C^k\), respectively. Hence, \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\) will not exist in the final output given of the algorithm which is given by \(C\).

In case I, we have shown that if a transition (in particular \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\)) is not in \(C^0\) then it will not exist in \(C\). Also, we have shown in case II from item (a) to item (f) that a transition (in particular \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\)) is not in \(C^k\) if that transition violates one of the following constraints: (i) controllability (ii) nonblocking (iii) communication–errors constraints (iv) marked as disabled. Hence,
if a transition (in particular \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\)) is not in \(C\) which is the final output of Algorithm 3.1, then it is because it does not satisfy one of the above constraints since \(C\) is constructed from \(C^k\).

It follows that when \(C\) is coupled with \(\mathcal{G}_W\) (i.e., \(C \otimes \mathcal{G}_W\)), if a transition (in particular \((s_n, t_n) \xrightarrow{\delta_n[g_m]} (s_{n+1}, t_{n+1})\)) is not in the coupled system \(C \otimes \mathcal{G}_W\), it is due to one of the following reasons: (i) the transition does not satisfy the definition of controllability, (ii) the transition leads to blocking, (iii) the transition results in communication errors, or (iv) the transition is marked disabled in \(C\) and thus, disabled in \(C \otimes \mathcal{G}_W\) because the transition is controllable. Therefore, as stated in (iv), we conclude by contradiction that \((c'_n, s_n) \xrightarrow{\delta_n[g_m]} (c'_{n+1}, s_{n+1}) \notin \Gamma_{C'}\). Since, if this transition \(((c'_n, s_n) \xrightarrow{\delta_n[g_m]} (c'_{n+1}, s_{n+1}))\) is in \(C'\), then \(C'\) will not satisfy one of the above reasons. That is, Algorithm 3.1 is designed to satisfy these conditions. ■

A direct consequence of Theorem 3.5.2 is that the controller generated is communication-error free and nonblocking, which we state in the following proposition and theorem without proofs.

**Proposition 3.5.1** (communication-error free). The controller \(C\) generated by Algorithm 3.1 is communication-error free.

**Theorem 3.5.3** (safe and nonblocking). The controller \(C\) generated by algorithm 3.1 is a nonblocking and safe controller.

**Theorem 3.5.4** (Termination). Algorithm 1 converges in a finite number of iterations.
Proof. To prove that Algorithm 3.1 terminates we first show that each of the other algorithms (i.e., Algorithms 3.2 to 3.10) called by Algorithm 3.1 terminate and finally we show that Algorithm 3.1 terminates. We show that every loop and recursive call terminates by providing the necessary termination arguments.

(i) Algorithm 3.2

The procedure in Line 17 of this algorithm performs a fixed-point iteration of $\mathcal{G}_W$ to remove communication errors. The for loops starting at Line 22 and Line 23 iterate over the states and output transitions of $\mathcal{G}_W$, respectively. The states and output transitions of $\mathcal{G}_W$ are finite. Thus, these loops terminate.

More importantly, in Line 25 the algorithm calls the procedure findAllPath defined in Lines 3–16. The procedure findAllPath recursively finds all possible paths from a given state with each state and each transition visited at most once in each path. This avoids loops, which guarantees the termination of the findAllPath procedure. Lines 26–28 of the algorithm rely on the output of the findAllPath procedure to eliminate transitions which result in unspecified receptions. Lines 26–28 of the algorithm are bound to terminate since the number of paths considered is finite. Thus, the loops starting at Line 22 and Line 23 of the algorithm terminate in a finite number of steps. A similar termination argument can be derived for Lines 35–45 of this algorithm. Now, the while loop starting at Line 20 is bound to terminate since either all communication errors are eliminated and $\mathcal{G}_W$ remains the same after a certain number of iterations or $\mathcal{G}_W$ becomes empty.

(ii) Algorithm 3.3
The outer loop in Line 4 loops over the states of the plant $G_W$ and the specification $T_W$, respectively. This loop terminates since the set of states of $G_W$ and that of $T_W$ are finite. The two inner loops in Line 5 and Line 8 iterate over the set of static and dynamic transitions, respectively. Since these sets are finite it implies that these two inner loops terminate. Similarly, the loop in Line 21 also terminates since it is bounded by the size of static transitions in $G_W$.

(iii) **Algorithm 3.4**

This algorithm has three main for loops. The first for loop searches for all states and transitions that result in blocking and collects them into a buffer. It terminates when all transitions and states leading to blocking have been collected. The second nested for loop iterates over the set of states and transition of $C^k$ and eliminates all blocking states and transitions collected in the previous steps. This loop terminates when all blocking states have been eliminated from the states of $C^k$. The last for loop in this function creates new states to preserve the structure of dynamic transitions and terminates when all new states have been created.

(iv) **Algorithm 3.5**

In this algorithm, the iterations in Lines 18–34 perform a search on the set of states and transitions of $C^k$ and collect all the states and transitions that lead to bad states based on whether the value of a given variable can be modified or not from when it was declared to when it was used. Since the set of states and associated transitions of $C^k$ is finite, the iterations will eventually terminate.
This is because there will be a step in the iteration in Lines 18–34 where there will be no more transitions that lead to bad states to collect.

In the iteration in Lines 38–46, there are four main \texttt{for} loops. The outermost loop (Line 38) loops over the set of bad state BS collected from previous step and then the second outer loop iterates over the set of transitions that leads to bad states also collected in the previous states. The two innermost loops search for the guards of the transitions that needs to be strengthened and then in Line 42 the algorithm strengthens these guards. The iteration terminates when all transitions in BS have been strengthened.

Lines 49–61 of this algorithm have three loops which search for all the set of variables that must be monitored at runtime and collect them into an array called \textit{runtimeVariable}. These iterations terminate when all the variables to be monitored at runtime have been collected and stored.

Finally, Line 62 of Algorithm 3.5 also calls the function \texttt{disableControllableDynamicTransition} which iterates over the set of transitions of $C^k$ and disables all controllable transitions which are enabled in $G_W$ but are not allowed in $C^k$ because the guards on both transition are not satisfied. Here, termination is guaranteed since there exists a finite number of controllable transitions.

(v) \textbf{Algorithm 3.6}

The inner loop of this algorithm is bounded by the size of $\mathcal{S}_C$ and the outer loop is bounded by the size of $\mathcal{S}$. Thus, after each iteration of the outer loop
the inner loop terminates after $|S_{c}|$ (where $|S_{c}|$ is the cardinality of $S_{c}$) steps and after $|S|$ (where $|S|$ is the cardinality of $S$) steps the outer loop eventually terminates.

(vi) **Algorithm 3.7**

In Algorithm 3.7 the set of bad states $S_{Ck}^{Bad}$ is finite. Now, in the while loop in Lines 2–7 any transition that results in a bad state is iteratively removed from $C$ as well as the bad state itself. Now, when all the transitions leading to the bad state have been removed, $S_{Ck}^{Bad}$ will be empty which terminates the while loop of this algorithm. Thus, Algorithm 3.7 terminates.

(vii) **Algorithm 3.8**

The proof of termination of this algorithm is trivial, since both for loops in Lines 2–6 and Lines 7–11 are bounded by the number of reachable states and transitions in $C$ (i.e., the input of Algorithm 3.8).

(viii) **Algorithm 3.9**

The for loop in Lines 6–22 iterates over each pair of $(t_{i}, s_{i}) \in R$. This for loop is bounded by the size of $R$. Since $R$ is finite, it implies that this for loop will eventually terminate. Now, for each iteration of the for loop in Lines 6–22 a transition in $T^{W}$ is either eliminated (Line 10 and 17) or the guard on this transition is changed to that of the plant $G_{W}$. After repeated execution of this for loop, there will be a point where all the transitions that cannot be matched by the plant have been eliminated or the guard on those transitions have been changed to that of $G_{W}$ (and in the worst case $T^{W}$ will
become empty). Thus, $T_W$ will remain unchanged at this point which implies that in Line 23 $T_W' == T_W$ and the repeat until loop terminates.

(ix) Algorithm 3.10

This algorithm implements two main loops. The inner loop in Lines 12–17 loops over the set of transitions at each state $s$ satisfying the if condition in Line 7. Since the number of transitions at each state $s$ is finite, it implies that this loop will eventually terminate once all the transitions at each state have been analyzed. In addition, the outer loop starting at Line 6 iterates over the set of states $S_M$ of $M$ (i.e., the input to Algorithm 3.10) and given that the size of $S_M$ is finite, the outer loop terminates in a finite number of steps. Hence, Algorithm 3.10 terminates.

Finally, we are ready to show that Algorithm 3.1 terminates. Since, we have already shown that all the other algorithms called by Algorithm 3.1 terminate, we have left to show that the repeat until loop in Lines 8–13 terminates too. Now, since for every iteration of the repeat until loop (a) Line 10 checks the static controllability condition on some $C^k$ and removes some states and transitions, it also adds certain bad states and transitions; (b) Line 11 checks for blocking conditions and eliminates blocking states and associated transitions and (c) Line 12 checks for the dynamic controllability conditions and strengthened guards that do not satisfy these conditions as well as removing states and transitions which do not satisfy the second item of Definition 3.3.8. After a finite number of iterations, $C^k$ is bound to remain unchanged as it verifies the above conditions (a), (b) and (c). Thus, we will eventually have $C^k = C^{k-1}$ which terminates the repeat until loop in Line 13. Therefore, we
3.6. ANALYZING THE COMPUTATIONAL COMPLEXITY

conclude that Algorithm 3.1 terminates in a finite number of steps. ■

3.6 Analyzing the Computational Complexity

Here, we present a brief analysis of the computational complexity of the relevant part of Algorithm 3.1. Recall that the initial inputs to Algorithm 3.1 are the asynchronous parallel composition of all the available services given by $G_W = (S^W, S^0_W, I^W, O^W, A^W, \Gamma^W, S^F_W, V^W, B^W)$ and the composition requirement represented as $T^W = (S^W, S^0_W, I^W, O^W, A^W, \Gamma^W, S^F_W, V^W, B^W)$. Let $|X|$ denote cardinality of a given set $X$. We recall that the number of reachable states of the SLTS $G_W$ is finite and also the number of variables in $G_W$ over the domain $D_G^W$ is finite (fixed). Now, assume that each variable $v(i)$ such that $i = 1, 2, 3, \ldots, k$ in $V^W$ can take up to $k$ finite values in the domain $D_G^W$ of the set of variables, then the maximum number of possible values in $D_G^W$ of all variables is given by $|D_G^W| = \prod_{i=1}^{k} |v(i)|$, where $|v(i)|$ is the finite possible values the variable $v(i)$ can take. Thus, the number of possible transitions in $G_W$ is bounded by $|X| = |S^W||D_G^W|$. Similarly, the number of possible transitions in $T^W$ is bounded by $|Y| = |S^W||D^W|$. Therefore, the number of reachable states and transitions of the input to Algorithm 3.1 is bounded by $|X||Y|$. We assume that all states in $S^W$ are reachable. By Lemma 3.5.4, we know that the set of states and transitions of $C^0$ cannot exceed that of $G_W$. Therefore, the number of reachable states in $C^0$ is bounded by $|S^W|$ and the number of all possible transitions in $C^0$ is also bounded by $|X|$.

In what follows, we only present the complexities of the main loop (repeat until in Lines 8–13) of Algorithm 3.1. The complexities of the other parts of the algorithm
do not affect the results presented here.

- **Lines 8–13 of Algorithm 3.1:** The main loop of Algorithm 3.1 loops over $C^0$. In every iteration of this loop some states of $C^0$ and the associated transitions are eliminated or some guards of certain transitions are strengthened if the valuation of these guards result in bad states. In the worst case, this `repeat until` loop will iterate $O(|X|)$ times until the number of good states and transitions leading to good states of $C^0$ become empty. Therefore, the complexity of the `repeat until` loop is given by

$$O(|X|)$$  

(3.10)

Although, new bad states are created in the process, these states do not affect the loop, since if $C$ is made of only bad states the loop will terminate after a single execution.

- **Line 10 of Algorithm 3.1:** This line of the algorithm calls Algorithm 3.3 to perform static controllability. Now, the first loop starting from Line 4 of Algorithm 3.3 iterates over the states of $G_W$ and the states of $C^k$, respectively. The complexity of this loop is $O(|S_{G_W}|^2)$. The next loop in Line 5 loops over the set of static transitions of $G_W$. In the worst case the set of static transitions of $G_W$ cannot be larger than the set of all possible transitions of $G_W$. Thus, we take $O(|X|)$ as the complexity of this loop. Similarly, the loop in Lines 8–13 loops over the states and the dynamic transitions of $C^k$ which gives a complexity of $O(|X|)$. All other lines of Algorithm 3.3 have complexity of $O(1)$ each. In a
3.6. ANALYZING THE COMPUTATIONAL COMPLEXITY

In nutshell, the complexity of Algorithm 3.3 is determined by the first three loops which is

\[ O(|S_W|^2|X|^2) \]  \hspace{1cm} (3.11)

- **Line 11 of Algorithm 3.1:** This line of Algorithm 3.1 is implemented by Algorithm 3.4. Algorithm 3.4 has three main outer loops, (i) the first loop in Lines 8–12 loops over the states of \( C^k \) which gives us a complexity of \( O(|S_W|) \); (ii) the second loop in Lines 14–19 iterates over the set of good states and the dynamic transitions of these good states of \( C^k \). In the worst case the set of good states and their associated transitions in \( C^k \) cannot be larger than \( |X| \) which consists of both bad and good states. Thus, the complexity of this loop is \( O(|X|) \); and (iii) the outer loop starting at Line 22 iterates over the set \text{BlockandUnsafe}. In the worst case \text{BlockandUnsafe} cannot be bigger than the set of states of \( C^k \) itself. Hence, the complexity of this loop is given by \( O(|S_W|) \). The inner loop starting at Line 23 loops over the set of all possible transitions of \( C^k \), which implies that the complexity of this loop is given by \( O(|X|) \). Therefore, the combined complexity of the two loops in Line 22 and Line 23 is \( O(|S_W||X|) \). From (i)–(ii), we determine the complexity of Algorithm 3.4 to be

\[ O(|S_W||X|) \]  \hspace{1cm} (3.12)

- **Line 12 of Algorithm 3.1:** To compute dynamic controllability, Algorithm 3.1 calls upon Algorithm 3.5 in Line 12. Now, we determine the computational complexity of Algorithm 3.5 as follows:
(i) Lines 18–34: The outer loop starting from Line 18 has complexity of $O(|S_{gw}|)$ since it iterates over the states of $C^k$. On the other hand, the inner loop in Line 19 iterates over the set of reachable states and all possible transitions of $C^k$ which gives us a complexity of $O(|X|)$. In total, the two loops from Lines 18–34 have a complexity of $O(|S_{gw}|.|X|)$.

(ii) Lines 38–46: The outermost loop of these lines loops over the bad states of $C^k$. In the worst case the set of bad states cannot be larger than the entire state set of $C^k$. Thus, the complexity of the outermost loop starting at Line 38 is $O(|S_{gw}|)$. The next two inner loops starting at Line 39 and Line 40 have complexity of $O(|X|)$ each. Now, we estimate the complexity of the innermost loop as follows. Let $np$ denotes the maximum number of paths starting from the initial state in $C^k$ such that for each path a state is visited once (i.e., it avoid loops). The innermost loop starting at Line 41 iterates over the set of paths of $C^k$ such that each path in this set starts from the initial state and has a specific input transition $t$ in it. In the worst case the maximum number of iterations of this innermost loop cannot exceed the number of paths in $C^k$. Thus, we take $O(np)$ as the complexity of this loop. Combining all the results for the four loops in Lines 38–46 gives $O(np(|S_{gw}|.|X|^2))$.

(iii) Lines 49–61: The loop beginning in Line 49 has complexity of $O(|S_{gw}|)$ since in the worst case the elements in the set $BPN$ cannot exceed the entire state set of $C^k$. Also, the loop starting at Line 61 has complexity of $O(|X|)$ in the worst case. Summing up the complexity of the loops in
Lines 49–61 we get $O(|S_{gw}||X|)$.

From (i)–(iii) we calculate the complexity of Algorithm 3.5 to be

$$O(|S_{gw}||X|) + O(np(|S_{gw}||X|^2)) + O(|S_{gw}||X|)$$

which can be simplified by retaining the term with the highest exponent to get

$$O(np(|S_{gw}||X|^2)) \quad (3.13)$$

Finally, summing up Equations 3.10, 3.11, 3.12 and 3.13 we arrive at

$$O(|X|)(O(|S_{gw}|^2|X|^2) + O(|S_{gw}||X|) + O(np(|S_{gw}||X|^2))) \quad (3.14)$$

and retaining the terms with the highest exponent we have

$$O(|X|)(O(|S_{gw}|^2|X|^2) + O(np(|S_{gw}||X|^2))) \quad (3.15)$$

which can be simplified to get

$$O(|X|^3|S_{gw}|(|S_{gw}| + np)) \quad (3.16)$$

as the complexity of the main loop of Algorithm 3.1 which determines the complexity of the entire algorithm.
Chapter 4

SASCA Prototype Implementation

In this chapter, we introduce the system which implements the proposed approach. First, we present an overview of the architecture of the system and a synopsis of its implementation details. Second, we demonstrate how the system works using Example 3.4.1. In Chapter 5, we present an evaluation of this tool and the approach as a whole using case studies inspired by business applications.

4.1 SASCA Architecture

This dissertation proposes a new framework called SASCA (Supervisory Aware Service Composition Architecture) for modeling Web service composition. The SASCA framework provides an implementation of the formal framework presented in the preceding chapter (Chapter 3). It consists of (a) a translator which converts a given abstract WS-BPEL specification to its SLTS equivalent, and vice versa; (b) a composition generator which relies on supervisory control theory to synthesize a provably correct-by-construction controller; (c) an augmentation of the constructed controller
to handle runtime information to prevent runtime failure; and (d) a refinement module to assist developers to modify a specification.

Figure 4.1 depicts the basic architecture diagram for the SASCA framework. The inputs to the system are the set of component Web services specified as abstract WS-BPEL or as SLTS and the composition requirements specified as SLTSs. The SASCA software tool allows the composition of the component Web services in either their abstract WS-BPEL or SLTS forms. This feature allows a developer to reuse existing services that are already published and specified in WS-BPEL. However, the specification must be given in SLTS form. In the case that the component Web services are supplied to the system in their WS-BPEL representations, a built-in translator of the SASCA framework is used to generate the corresponding SLTSs representations from the WS-BPEL descriptions of these services. The diagram shows the important internal representations from when the inputs enter the system to when a controller is generated. The diagram also depicts an intermediate preprocessing step of the plant and the specification which removes any communication design errors to achieve a more refined model suitable for composition synthesis. More specifically, there are two stages involved which are the preliminary stage and the composition stage. In the preliminary stage, given the component Web services $G_{W_1}, G_{W_2}, \ldots, G_{W_n}$, the system combines these component services together using the asynchronous parallel composition to produce the plant $G_W$. Next, communication design errors are removed (Ref($G_W$)). In the composition stage, the SASCA system first checks that the plant simulates the specification. Once the simulation relation check is successful, the generation of the controller and the augmentation of the controller to enforce
Figure 4.1: Supervisor Aware Service Composition Architecture (SASCA)
control using runtime information is performed. The final output of the synthesis process is a set of WS-BPEL executable files which can be exported into any WS-BPEL execution engine for deployment. We show how to deploy the executable files in Chapter 5. The tool also provides various test units for testing purposes. In the rest of this chapter, we discuss the implementation details as well as how to execute the toolkit.

4.2 Prototype Implementation

The SASCA framework has been developed solely in the Java programming language in order to make the tool platform-independent and to take advantage of the benefits offered by the object-oriented programming paradigm. To specify and read an SLTS from a given file, a parser was developed using state-of-the-art compiler technologies. The parser was automatically generated using an LALR parser generator called Java Cup [78] and a lexical analyzer generator known as JFlex [91]. To do this, a grammar and a specification with a set of regular expressions and corresponding actions as inputs are supplied to Java Cup and JFlex, respectively. Given an SLTS specified in a file as shown in Listing 4.1, we use Java Cup and JFlex to generate an abstract syntax tree (Java objects) which we traverse to derive the SLTS. The next section describes how an SLTS is represented internally in the system.
### 4.2. PROTOTYPE IMPLEMENTATION

#### Listing 4.1 SASCA SLTS specification of the Plant Services

```plaintext
WEBSERVICE
  [Plant_Services];

[STATES] {<S0, S1, S2, S3, S4, S5, S6, S7>};
[STARTSTATE] {<S0>};
[FINALSTATE] {<S0, S5, S6>};

[TYPES] {Int <Integer>, Data <String>};

[ VARIABLES ]
  {x <Int>, y <Data>, z <Int>,
   v <Int>, z <Data>, var <Data>};

[EVENTS] {
  [INPUTS] {<msg1(x), msg2(z), msg3(var), fail_f()>};
  [OUTPUTS] {<msg1(x), msg2(z); msg3(var); fail_f()>};
  [ATOMICOPERATIONS] {
    <atom1(x::y), atom2(y::v), atom3(x)>};

  [TRANSITIONS] {
    {<S0 => ?fail_f() => S0>};
    {<S0 => !msg1(x) => S1>};
    {<S0 => !msg3(var) => S1>};
    {<S1 => ? msg1(x) => S2>};
    {<S1 => ? msg3(var) [(var = cr) || (var = dr)] => S7>};
    {<S2 => atom1(x, y)[x<7] => S4>};
    {<S3 => ? msg2(z) => S1>};
    {<S3 => ? msg1(x) => S5>};
    {<S4 => ! fail_f() => S0>};
    {<S4 => atom2(y, v) [(y=a) || (y=b)] => S0>};
    {<S5 => atom3(x) => S2>};
    {<S5 => msg2(z) => S6>};
    {<S7 => ! msg2(z) => S3>};
  }
}
```

#### Listing 4.2 SASCA SLTS specification for the requirements

```plaintext
WEBSERVICE
  [Goal_Service];

[STATES] {<T0, T1, T2, T3, T4, T5, T7>};
[STARTSTATE] {<T0>};
[FINALSTATE] {<T0, T5>};

[TYPES] {Int <Integer>, Data <String>};

[ VARIABLES ]
  {x <Int>, y <Data>, z <Int>,
   v <Int>, z <Data>, var <Data>};

[EVENTS] {
  [INPUTS] {<msg1(x), msg2(z), msg3(var), fail_f()>};
  [OUTPUTS] {<msg1(x), msg2(z); msg3(var); fail_f()>};
  [ATOMICOPERATIONS] {
    <atom1(x::y), atom2(y::v), atom3(x)>};

  [TRANSITIONS] {
    {<T0 => ?fail_f() => T0>};
    {<T0 => !msg1(x) => T1>};
    {<T0 => !msg3(var) => T1>};
    {<T1 => ? msg1(x) => T2>};
    {<T1 => ? msg3(var) [(var = cr)] => T7>};
    {<T2 => atom1(x, y)[x<3] => T4>};
    {<T3 => ? msg2(z) => T1>};
    {<T3 => ? msg1(x) => T5>};
    {<T4 => ! fail_f() => T0>};
    {<T4 => atom2(y, v) [(y = b)] => T0>};
    {<T5 => atom3(x) => T2>};
    {<T7 => ! msg2(z) => T3>};
  }
}
```

### 4.2.1 Representing Services in SASCA

An SLTS is represented by a root class called `SLTS_Automata` class. The `SLTS_Automata` class comprises all the parts of an SLTS defined as attributes and provides various methods to manipulate the SLTS. The UML class diagram including relevant interfaces of the `SLTS_Automata` class is shown in Figure 4.2. This class consists of
the following attributes:

(i) **name**: This attribute represents the name of the SLTS_Automata, which is a unique identifier used to retrieve and refer to the SLTS_Automata class.

(ii) **setOfStates**: This attribute represents the set of states of the SLTS_Automata. A state is defined by a number of attributes, namely, the name of the state `state_name`, the unique identifier `state_ID` used to refer to the state, a Boolean identifier `isFinal` to indicate whether a state is final or not, and finally, a state vector called `stateProductVector` is used to represent a state name that has more than one dimension, i.e., a pair of states of the form \((p, q)\).

(iii) **initialStateSet**: This attribute corresponds to the initial state of the SLTS_Automata.

(iv) **finalStateSet**: This attribute denotes the set of final states of the SLTS_Automata.

(v) **variableSet**: This attribute denotes the set of variables in the SLTS_Automata. A variable consists of a `name` and a `type`.

(vi) **eventsSet**: This attribute represents the set of actions of the SLTS_Automata, the set of actions are made up of the atomic actions, the input messages and the output messages. These actions in turn have either a set of input variables (input message), output variables (output message) or both (atomic actions).

(vii) **types**: The representation of the SLTS_Automata allows the user to declare the type of a variable as a primitive Java type or a complex type which specifies
the user’s own type.

(viii) **partnerLink**: This attribute is used to represent the `partnerLinkType` of a WS-BPEL specification in the case that the `SLTS_Automata` is derived from WS-BPEL.

(ix) **transitionSet**: This attribute denotes the set of transitions of the `SLTS_Automata`. A transition consists of multiple attributes which are given by (a) the `name` of the transition; (b) the `beginState` and the `endState` which denote the start and end state of the transition, respectively; (c) the event name `action_Name` which could be an atomic action, an input or output message; (d) the guard on the transition represented by the attribute `condition`. To aid the evaluation of guards, we used a version of a tool for representing and evaluating first order logic formulas [51]. Table 4.1 shows the predicate operators that the current implementation of SASCA can recognize and evaluate in a guard of a given transition; (e) a number of flags such as `isStatic`, `isDynamic`, `isForcible` and `isControllable` which flags whether a transition is static, dynamic, forcible or controllable, respectively.

The SASCA framework carries a module that transforms a given SLTS into an equivalent Graphviz [153] dot representation. This aids the user during the service composition process to view and edit SLTSs graphically using the Graphviz graphical viewer tool.
### 4.2.2 The SASCA Translator

In this section, we give the relevant details of the SASCA translator. To build a translator that translates a WS-BPEL process and its associated WSDL files (parnerLinkTypes) into a corresponding SLTS and vice versa, we developed an XML parser for parsing the WS-BPEL process, including its associated WSDL files. The translation process involves the manipulation and handling of a large amount of XML data. To do this translation, we used a Java API called Java Architecture for XML Binding (JAXB) [121] to map the XML documents into Java objects and vice versa. More specifically, the JAXB tool was used to generate Java objects from the
Figure 4.2: Class Diagram of SLTS_Automata
abstract WS-BPEL XML schema [5] and the WSDL XML schema [81] files, respectively. Given the abstract syntax tree of these Java objects, we traverse it to retrieve and store the relevant data elements. This stored data is then transformed into an equivalent SLTS. To translate a WS-BPEL process into an SLTS, one must provide the WS-BPEL process file and the WSDL file as inputs to the translator. The translator first validates these inputs against the WS-BPEL XML schema and the WSDL XML schema, respectively. Once the validation is successful, the translator extracts the data and then uses the transformation rules given in Section 4.2.3 to generate an SLTS. Conversely, given an SLTS, the translator can systematically generate a corresponding executable WS-BPEL process and its associated WSDL files. The current implementation of the translator deals with the most relevant subset of the WS-BPEL language given in Table 3.2. In more detail, the basic and structured WS-BPEL activities that the translator supports are receive, reply, invoke, sequences, assign activity, while, pick, exit, if, empty and sequence activities. We only discuss the translation of the most relevant activities in this dissertation.

Existing work [67, 103] has discussed techniques to translate a WS-BPEL into an automaton. However, our approach involves more complex transformation rules that are built on communicating services and manipulates XML data. In fact, there is no existing approach that provides a translator which translates a WS-BPEL process to a Labelled Transition system augmented with guards and data variables. In addition, most of the existing approaches are based on WS-BPEL 1.0 [6], whereas the translation technique present here is based on WS-BPEL 2.0 [5]; even so, these existing approaches do not make their translators available for public use.
4.2.3 Transformation Rules

In this section, we describe the relevant transformation rules used for translating a WS-BPEL process into a corresponding SLTS and vice versa. That is, we present the details of translating a WS-BPEL to an SLTS and back, concurrently. It is important to note that not all the WS-BPEL constructs are translated in the current implementation of the translator. For example, advanced WS-BPEL constructs like faultHandlers and compensationHandler are not handled. In the following, we present how the relevant WS-BPEL constructs were taken care of.

- **partnerLinkType**: The WS-BPEL partnerLinkTypes defines the relationship between partner Web services (imported WSDL files) using roles. Each role is associated with a portType through which a WS-BPEL process can refer to the operations and types defined in the WSDL file. To accommodate partnerLinkTypes in the SLTS, a data structure is created to store all information relating to each partnerLinkType defined in the WS-BPEL process. This data structure is attached to the SLTS which enables us to refer to the elements such as types, operations and messages defined in the WSDL file of a particular partnerLinktype. This is also useful when translating the SLTS back into its corresponding WS-BPEL process.

- **WSDL types**: A WS-BPEL process uses the types defined in the WSDL files of its partnerLinkTypes. These types could be XML primitive types or user defined types. These types are mapped into the types of an SLTS and vice versa during the translation process.
4.2. PROTOTYPE IMPLEMENTATION

- **variables**: The WS-BPEL process supports variables which could be WSDL Message types or XML Schema types. The WSDL Message types’ variables correspond to Web service message types that are defined in the WSDL files and also imported by the WS-BPEL process, whereas the XML Schema types’ variables correspond to simple or complex XML Schema data types. To translate variables, we map both the WSDL Message types and the XML Schema types variables into the set of variables of the SLTS. In the opposite way, during the translation of an SLTS back into a WS-BPEL process, the variables in the SLTS are mapped into the variables of the WS-BPEL process.

- **sequence activity**: The sequence activity is first translated into two states of an SLTS and then its sub-activities are systematically translated and placed between these two states.

- **reply activity**: The reply activity of the WS-BPEL process is translated into an output transition of the corresponding SLTS such that the operation of the reply activity corresponds to the action labeling the transition and the variables of the reply activity are mapped into the set of output variables of this action, respectively. In the reverse case, an output transition of an SLTS is translated into a reply activity of the WS-BPEL process (see Figure 4.3).

- **receive activity**: Similar to the reply activity, the receive activity of WS-BPEL process is translated into an input transition of the corresponding SLTS where the operation of the receive activity becomes the action labeling the transition. The variables of the receive activity are mapped into the set
4.2. PROTOTYPE IMPLEMENTATION

Figure 4.3: Translation of reply Activity into Output Transition

of input variables of the transition. Similarly, in the reverse case, an input
transition of an SLTS is translated into a receive activity of the WS-BPEL
process (see Figure 4.4).

Figure 4.4: Translation of receive Activity into Input Transition

- invoke activity: The invoke activity of the WS-BPEL process is translated
into an atomic operation transition of the corresponding SLTS. The operation
on the invoke activity is mapped into the action that labels this atomic operation transition, whereas the input and output variables of the invoke activity
are translated into the input and output variables of the action of this atomic operation transition of the SLTS, respectively. A similar transformation is used
for translating an atomic operation of an SLTS derived from an invoke activity
into a WS-BPEL process invoke activity (see Figure 4.5).
4.2. PROTOTYPE IMPLEMENTATION

Figure 4.5: Translation of invoke Activity into Atomic Operation Transition

- **assign** activity: The **assign** activity of the WS-BPEL process is translated into an atomic operation transition of the SLTS. In this case, the **from** and **to** part of the **copy** operation in the **assign** activity are converted into the input and output variables of the action on the transition, respectively (see Figure 4.6). In the current implementation, there is a limitation of which expressions can be used in the from part and to part of the **assign** activity.

Figure 4.6: Translation of assign Activity into Atomic Operation Transition

- **while** activity: The **while** activity allows for repeated execution of the contained activity. It is made up of a Boolean condition and a loop which is performed as long as the Boolean condition evaluates to true at the beginning of each iteration. To translate the **while** activity into a corresponding SLTS transition, first, we translate the body of the loop which starts and ends in the same state (loop). The positive Boolean condition (true) of the **while** activity
is attached as a guard to each transition in the loop body. Next, the activity preceding the **while** activity upon exit is also translated into a corresponding SLTS transition and the negative Boolean condition (false) is attached to this transition. Consider the **while** activity of the WS-BPEL code excerpt in Figure 4.7(a), the body of the loop (Lines 3-13) is translated into the transitions from state $S_1$ to $S_2$ and $S_2$ to $S_1$, respectively of Figure 4.7(b). Next, the **invoke** activity in Lines 15-18 is translated into the transition from $S_1$ to $S_3$. Finally, the positive part of the loop condition $status > 0$ is attached to the transitions from state $S_1$ to $S_2$ and $S_2$ to $S_1$, while the negative part of the Boolean condition $status \leq 0$ is attached to the transition from $S_1$ to $S_3$, accordingly.

(a) **while** Activity

(b) **while** Condition Transition

![Figure 4.7: Translation of **while** Activity](image)

- **if** activity: The **if** activity is used to define different behaviours of the process by using conditional behaviour to decide between two or more branches of
execution. Each condition of a branch is in the form of an XPath expression [37]. In the case that the expression is true, then the branch is executed. If the expression is false, then the next valid branch is executed. Thus, to translate the if activity into an SLTS, we translate each branch separately, such that the positive condition represents the if branch and the negative condition represents the else branch of the SLTS accordingly. The if activity also comes with an if else branch. This is also translated by adding an additional branch to the if transition to accommodate it. The XPath expression of each branch of the if activity is used as the guard for the corresponding transition in the SLTS. The SLTS in Figure 4.8(b) represents the translation of the code excerpt in Figure 4.8(a). A similar mapping rule is used to translate multiple branches of an SLTS into an if activity of a WS-BPEL.
4.2. PROTOTYPE IMPLEMENTATION

```xml
<if><condition>$status>10</condition>
<receive partnerLink="KLMAirlines" portType="FlightStatusPT" operation="FlightBooking" variable="FlightInfor" createInstance="yes" />
<elseif><condition>$0<status<=10</condition>
<assign name="assign">
<copy>
<from variable="FlightInfor" part="in"/>
<to variable="FlightVar" part="out"/>
</copy>
</assign>
</elseif>
<else>
<reply partnerLink="DeltaAir" portType="AvailPt" operation="FlightAvail" inputVariable="Details" />
</else>
</if>

<invoke partnerLink="EthiopianAirLine" portType="FlightCallbackPT" operation="FlightCheck" inputVariable="replyFlight" />

(a) if Activity
(b) if Condition Transition

Figure 4.8: Translation of if Activity

- **pick activity**: The **pick** activity provides multiple branches, each one is associated with an event which triggers the branch to be executed. An event can be an **onMessage** type or an **onAlarm** type. The **onMessage** signifies the arrival of a new message, whereas the **onAlarm** is a timer-based alarm which is a condition for something to happen. To translate the **pick** activity, we translate each branch as an **if** activity such that the events become the conditions for the **if** activity. Figure 4.9 illustrates this. The reverse case during the translation of a **pick** transition of SLTS to WS-BPEL **pick** activity is similar.
4.2. PROTOTYPE IMPLEMENTATION

4.2.4 Algorithm Implementations and Executing SASCA

In this section, we discuss the implementation of Algorithm 3.1 and how to use the SASCA system by means of Example 3.4.1. All algorithms presented in Section 3.4 have been implemented as Java methods using the Eclipse IDE [54]. The Java source code snippet for Algorithm 3.1 is given in Listing A.1 of Appendix A. The class diagram in Figure 4.10 gives an overview of the relevant Java API classes that implement all parts of Algorithm 3.1. To use the SASCA system, a call must be made to several methods in order to generate a composition. The code excerpt in Listing A.2 illustrates how the SASCA system can be initialized and called to generate a composition.
4.2. PROTOTYPE IMPLEMENTATION

Figure 4.10: UML Class Diagram of the Composition_Controller_Synthesizer Class and the Relevant Associated Classes
To demonstrate how to use the tool, let us consider Example 3.4.1 of Chapter 3. First, we encode the plant and the specification as shown in the code Listing 4.1 and Listing 4.2, respectively. Next, we provide each of these inputs to the function `loadSLTSAuto()` by initializing and calling it appropriately in a `main` Java method as shown in Lines 3-27 of Listing A.2 in Appendix A. After loading the inputs, next, we pass these inputs to the composition generator class by calling the proper Java methods as shown in lines 29-41 of Listing A.2. Compiling and running this code in a Java main class generates a command line output as shown in the screenshot in Figure 4.11. The generated controller and the asynchronous parallel composition of the available services can be viewed using a dot graphical viewer. A snapshot of the controller SLTSs is shown in Figure 4.12. The generation of the controller for this example took about 1140 milliseconds of CPU time. This experiment was performed on a 64-bit Intel (R) i5 desktop computer with 12.0 GB of memory.
4.2. PROTOTYPE IMPLEMENTATION

Figure 4.11: Snapshot Running Example 3.4.1 in SASCA in Eclipse IDE
4.2. PROTOTYPE IMPLEMENTATION

Figure 4.12: The Generated Controller Snapshot Executing Example 3.4.1 in SASCA in Eclipse IDE
Chapter 5

Application and Evaluation

The objective of this chapter is to present the application and a preliminary evaluation of the proposed composition approach in terms of (i) its effectiveness for the generation of controllers for a composition problem, (ii) its applicability using two small case studies (i.e., how to model a service composition problem using our approach), and (iii) the computational complexities with respect to time and space needed to generate a controller. In the automated service composition research community, there are no known existing benchmark algorithms or public test sets that we can compare with our approach. Thus, we choose case studies and examples popularly used by the service composition research community to validate their research findings.

We start this chapter by discussing the configuration of the environment for the experiments and how we conducted them. Next, we present the modeling of two case studies and some experimental results using our technique. For each of these two examples, we also report the results of deploying and executing the generated controller and the available services in the Oracle WS-BPEL Process Manager Engine.
Finally, we present a discussion on the results.

5.1 Description of Experimental Setup

All experiments were performed on a desktop workstation with 8.02\text{GBRAM}, Intel(R) Xeon(R) CPU running at 2\times3.40 \text{GHz} with both Ubuntu 16.04.2 and Windows 7 operating system installed. To deploy and execute the output from the SASCA system, we have used the Oracle Business Process Management engine since the free version is readily available for non-commercial public use, and is more robust than other engines. Oracle WS-BPEL Process Manager is now a component of the Oracle SOA Suite, which includes JDeveloper WS-BPEL Designer, WebLogic Server, Oracle Database and the Repository Creation Utility. The version of the SOA Suite used for this experiment is 11.1.1.7.0. The evaluation process involved the following steps:

- Modeling the available services
- Modeling the specification
- Using our system to generate a controller
- Deploying and executing

For each case study, we experimented with different variations by modifying the original component services, increasing or reducing the number of component services, and modifying composition requirements and scenarios. This diversification in the same case study allows us to evaluate various aspects of our framework.
5.2 Case Study 1: Travel Reservation

Our first experiment was performed with a variant of the travel reservation case study that we presented in Section 3.1. This example involves four component services; the Flight service, the Hotel service, the Bank service and the Interface service. Figure 5.1 and Figure 5.2 show the graphical representations of the Flight service and Bank service, respectively. These diagrams were created using the Oracle JDeveloper WS-BPEL designer. The code snippets for the Flight service in WS-BPEL and WSDL are displayed in Listing B.1 and Listing B.2 of Appendix B, respectively. The Hotel and the Interface services are omitted here.

Let us have a closer look at the Flight service in WS-BPEL displayed in Figure 5.1. It accepts a booking request using the fRequest receive activity. The flight details of the client are specified in this activity using the variable FLDdata which is of WSDL Message type FLOfferMsg (see code excerpt Listing B.1). In the WSDL file of this service, FLOfferMsg type specifies the date, the location and the kind of airline (KLM, Delta or AirCanada) (see code excerpt Listing B.2). It uses the assign activity checkFL_avail to check for the flight availability. The flight availability operation includes checking for the kind of airline specified by the client (i.e., KLM, Delta or AirCanada). Once the check is successful, the offer is processed and sent to the client using the offer invoke activity. The client may choose to accept the offer, which results in the completion of the booking, or the client may decline the offer, which results in cancellation. Figure 5.3(a) and Figure 5.3(b) depict the translated SLTSs of the Flight and Bank component services generated by the SASCA translator, respectively.
Figure 5.1: Flight Service from Travel Reservation Case Study
Figure 5.2: Bank Service from Travel Reservation Case Study
Figure 5.3: Generated SLTS of the Flight and Bank Services
Modeling the Composition Requirements  The composition requirements we considered are presented in Figure 3.4. Each of these specifications is encoded as SLTSs (similar to Figure 4.2). Self-loops were added at each state to indicate that transitions not mentioned in the state are allowed. We recall that the composition requirements for this case study are:

(1) The Hotel service cannot be booked if the flight is not available.

(2) The Flight service should only process booking for either KLM or Delta airline but not AirCanda airline.

(3) The Bank service can only accept credit or debit card but not Mastercard as a method of payment.

Running SASCA  Given the available service and the specification, we provided these inputs to SASCA and executed it. A snapshot of a portion of the synthesized controller is displayed in Figure 5.4.
5.2. CASE STUDY 1: TRAVEL RESERVATION

Figure 5.4: Portion of Controller for Travel Reservation
Table 5.1: Experiment Results

Table 5.1 (second row from bottom) reports the information about the plant and the generated controller for the given component services and requirements. The synthesis time is the time taken to construct the controller excluding the time used in building the asynchronous parallel composition of the component services (i.e., the plant), and the composition refinement of the plant and the specification.

Deploying and Executing on the WS-BPEL Engine: Once the controller has been generated, we manually inspected the generated WS-BPEL file for referencing of variables and XPath expressions. This inspection is done to ensure that the referencing of variables and XPath expressions corresponds to that of the Oracle WS-BPEL engine, since the implementation of our translator did not take into account any Oracle WS-BPEL engine and JDeveloper specifics. We also adjusted the correlation set activities of the generated WS-BPEL process to enable asynchronous communication in the WS-BPEL engine. This adjustment must be done as the translation process does not handle correlation set activities. Next, we manually copied the executable files generated by SASCA to deploy on the Oracle WS-BPEL engine. Once we deployed it, we observed the interaction between the
controller and the available services by inspecting the flow trace generated by the Oracle WS-BPEL engine testing interface.

First, we tested the running process by providing the correct inputs that will lead to a successful travel reservation. In this scenario, the controller correctly interacted with the client and the available services to make the travel reservation. In some instances, the execution resulted in WS-BPEL faults. These faults are due to errors relating to Oracle WS-BPEL engine specific references, such as mismatches in variable names, correlation set activities or mismatches in XPath references, but not due to a violation of composition constraints. Figure 5.5 shows a portion of the snapshot of the generated flow trace for this scenario. Another scenario we tested was to try to book a flight and request an AirCanda airline (i.e., by setting the flightType variable to AirCanada). In this case, the controller accepted the request but did not proceed with the booking. This is because the request violates the composition constraints. At this stage, the controller should have emitted a message to the client to indicate that the AirCanda airline is not available for booking using the event enforcement mechanism, but because this mechanism has not been implemented in the Oracle WS-BPEL engine, the execution halted.

Similarly, we tried to make a travel reservation with Mastercard as the method of payment, but this reservation did not go through as it violates the composition requirements. Payment with Mastercard is disabled from the beginning. This is because the controller foresees the danger of entering into an unsafe state and if this payment is allowed from the onset, it may result in unavoidable violations of the composition requirement.
In another scenario, we allowed the component services to interact directly with the client without the controller. We observed that there were instances where the Bank service had finalized its booking, but the Flight service had not been booked. With the controller, this situation is not possible.

We also tried to find out whether there are additional overheads of placing the controller between the client and the available services. We studied the response times of the controller between the times it receives a request and returns a reply. The experiment shows that this overhead is negligible. Hence, our approach does not incur additional overheads in this case.

5.3 Case Study 2: Virtual Heavy Duty Industrial Spare Parts Delivery Services

The next case study we considered is a variant of the example in the OASIS WS-BPEL standard [5, 6]. This case study is about an e-warehouse service composition organization, which we call Virtual Heavy Duty Industrial Spare Parts Delivery Services (VHDIS). This organization specializes in providing vehicular parts (spare parts, for example, Automobiles, Avionics) purchases and delivery services to satisfy a client’s request worldwide. All interactions are managed by Web services. The company’s objective is achieved by composing four or more of the following Web services. The Checkout service, the Shipping service, the Bank service, the Manufacturer service and the Warehouse service (which is made up of a Wholesale service and a Retail service). These services are described using WS-BPEL. The main goal is to compose all these services so that the user can directly ask the combined service
5.3. CASE STUDY 2: VIRTUAL HEAVY DUTY INDUSTRIAL SPARE PARTS DELIVERY SERVICES

Figure 5.5: Snapshot of Sequence of Interactions: Flow Trace Taken From the Oracle Business Process Manager (Enterprise Manager)
to purchase and deliver a given spare part.

- **Checkout service**: It serves as an interface through which the client can interact with the e-warehouse system. This component service receives inputs from the user and sends outputs to the user as well as to facilitate interactions among the available services (e.g., receiving a request from the customer to check the cost and the availability of an item).

- **Bank service**: This service checks and validates that a credit or debit card can be used to make a payment and transfers money from the client’s account to the e-warehouse’s bank account. This activity may fail or may be cancelled by the user.

- **Shipping service**: The Shipping service receives requests for transporting a product of a given size to a given location (the location can be LocA, LocB or LocC) if delivery is possible. The Shipping service provides a shipping offer with a cost and delivery time. This offer can be accepted or refused by the external service that invoked it. It provides the delivery status to both the user and other services involved. Figure 5.6 shows the WS-BPEL specification of the possible sequence of events that this service can take.

- **Manufacturer**: The warehouse service communicates with the manufacturer to replenish various products by providing the item’s name, the specification and the location of the warehouse. Then, it produces/assembles the item and ships it to a given warehouse.
- **Retail warehouse service**: This service processes retail purchases and receives a request to check the availability of an item. If a warehouse runs out of a product, it must request a restock from the manufacturer.

- **Wholesales warehouse service**: This service offers the same functionality as the retail service except that it processes wholesale purchases and provides the ability for customers to customize their spare parts.

Scenario: A typical scenario is as follows. (1) The customer’s point of contact is the front store (i.e., the Checkout service); (2) the customer places an order with the spare part’s specifications and the location to which it should be delivered; (3) the VHDIS reviews the order by checking the product availability, the cost, which warehouse(s) can supply the product, the shipping cost and time (if shipping is possible) as well as possibly requesting an emergency restock from the manufacturer, and finally sending a notification to the customer. The user can choose to accept or reject; (4) if the offer is accepted by the customer, VHDIS proceeds to check the customer’s credit card or authenticate his or her debit card and wait for the user’s confirmation of the offer; and (5) the VHDIS plans shipping, does the restocking/inventory and assembles the required product. At any point in time, the client has the option to cancel an order. It should be noted that not all of the services are required to participate in a composition, rather, some of them are selected as needed. For example, if a client’s service request is for a retail purchase, then there is no need to select the wholesale service as part of the composition.
Figure 5.6: WS-BPEL of the Shipping Service
Modeling the Composition Requirements: In this case study, we considered the following specification (stated informally):

(1) The Bank service must validate and check the credit rating of the customer, but cannot charge the customer’s credit card or debit card until the product is shipped.

(2) VHDIS should only ship spare parts to location LocB or LocC, but not LocA, that is, location ∈ {LocB, LocC}.

(3) VHDIS should only ship spare parts that are of class class1 or class3, but not class2, that is, class ∈ {class1, class3}.

Running SASCA: Given the component services of VHDIS and the composition requirement as inputs, we run SASCA over these inputs. Table 5.1 (last row) shows the number of participating services in the composition, the number of states and transitions of the asynchronous parallel composition of the participating services and the generated controller. The table also depicts the time in seconds used in generating the controller.

Deploying and Executing on the WS-BPEL Engine: The generated controller and the available services were deployed on the Oracle WS-BPEL Process Manager on the WebLogic Server after some manual inspection of the controller’s WS-BPEL executable file. Due to the large number of WS-BPEL activities in the generated executable file, we first deployed a small section of the generated WS-BPEL file and then tried to make it work on the server. Once the deployment was
successful, we increased the number of activities in the deployed file. We repeated these steps until all activities were added and deployed. The following observations were made by inspection of the flow trace generated by the engine:

The first test we conducted was to provide a correct set of inputs as a request to purchase a spare part from VHIDS retail services and execute it. The controller orchestrated the purchase activities of the available services without any violation of the specification.

Next, we tested the specification on event enforcement (see preceding page specification (3)). We placed a request to buy a spare part of class $\text{class2}$ by setting the spare part’s $\text{class}$ variable to $\text{class2}$. However, the controller did not know how to proceed as it can only process spare parts of $\text{class1}$ or $\text{class3}$. This transition is disabled on the controller transition system. Under normal circumstances, the controller should have triggered an enforceable transition to notify the client that a spare part of class $\text{class2}$ cannot be purchased, but this feature of runtime monitoring of variables and event enforcement has not been implemented in the Oracle WS-BPEL engine, which we leave for future work (it is a huge task and outside the scope of this thesis). A section of the snapshot of this instance of the execution is shown in Figure 5.7. In another setting, we tried to ship a product to location $\text{LocA}$. This was not possible as it was disabled at the controller’s transition. That is, the guard on the controller’s transition has been strengthened to exclude $\text{LocA}$ in the set of shipping locations since the controller anticipated a violation of the specification. Figure 5.8 displays the section of the flow trace generated by the Oracle Business Process Manager Service. Next, we tested the same scenario without the controller
and in some instances, a class2 spare part was processed, which violates the system requirements. In another instance, without the controller, we had the Bank service deduct money from the client’s account even though the purchase of the product had not been finalized yet. This violation did not happen when the controller was involved.
Figure 5.7: Snapshot of Sequences of Interactions: Flow Trace Taken From the Oracle Business Process Manager (Enterprise Manager)
5.4 Discussion

In this chapter, we have shown how our framework could be used to model real-world problems by means of the two case studies presented. Apart from these two case studies, we have also experimented with the well-known Loan Approval case study [5,6] and the results were not much different from what we have presented here.
The results show that the generated controllers prevent undesirable behaviours while otherwise restricting execution as little as possible. That is, the controller prevents a violation before it can occur. The result also demonstrates that the generated controller does not incur additional overhead when placed between the available services and the client. The results also highlight the fact that the service composition problem is exponential in the number of services in most cases [61, 114]. This claim became apparent in the experiment we conducted. More specifically, during the experiment, once the number of participating component services is six or more (and each of these services have at least five or more states), the SASCA prototype runs for days before terminating with an output. The implementation of the algorithms can be improved by using more efficient data structures and techniques, and when that is done, a better performance result with respect to execution time and space might be attainable. Most of the computational time was used to construct the composition refinement of the plant and the specification. In other words, with respect to the scalability of the proposed approach, the case studies we presented are relatively small ones, however with efficient implementation using advanced data structures as well as taking advantage of existing multi-core processors, the proposed approach should scale well with large real-world case-studies. In addition, there are a lot of manual steps involved in porting the generated controller to the WS-BPEL engine, which when dealt with can, go a long way towards improving the execution of the processes in the WS-BPEL engine. We leave the automation of these manual steps for future work.
Chapter 6
Conclusions

In this dissertation, we have developed a supervisory control framework for modeling Web service composition. The main contributions of this dissertation are as follows: We have developed a novel supervisory control framework for modeling the problem of automatic service composition in the SOA paradigm. The framework we have developed uses Labelled Transition Systems (LTSs) augmented with guards and data variables to model a given set of Web service specifications in industrial standard languages such as WS-BPEL. Our framework provides support for behavioural specification of services. More importantly, the approach we have proposed to deal with the problem of service composition is correct-by-construction. In the proposed framework, we have formalized the problem of automatic service composition as a supervisory control theory problem and have provided a formal solution to this problem, in which the generated supervisor have been proved to be correct. The novelty of this approach is the ability to enhance the generated controller, such that it is capable of enacting control based on information that is only available at runtime. Furthermore, we have established the correctness of our approach by formulating the
composition problem into theorems and providing proof of these theorems. First, we have proven the existence of a controller using our approach, by showing that given a set of available services $G_W$ and a goal service $T_W$, there exists a controller $C$ such that when the plant is coupled with this controller it satisfies the specification. Second, we have shown that the controller generated using our approach is minimally restrictive. Also, we have developed a set of algorithms to synthesize a composition using the formal framework.

We have extended DES theory so that processes represented by SLTSSs (which are a richer structure than FSMs) could be used to describe system behaviour. In particular, we have extended the standard SCT to handle important Web service constructs such as data, messages, variables, and guards. The services we model exhibit nondeterministic behaviours for the reason that some variable values are not known until runtime. This nondeterministic behaviour makes the research presented in this dissertation nontrivial and difficult. This situation could play out not just in the services world but in other settings in which some information of a variable does not become known until runtime, which means that the existing DES framework of Ramadge and Wonham does not suffice. To deal with this situation, we have integrated runtime input into the supervisor synthesis process such that the generated controller is able to enforce control based on the runtime information.

To demonstrate that the theoretical framework that we have proposed is implementable, we have developed a software prototype toolkit called SASCA which can be used to generate a controller for a given composition problem using our approach. We have used the proposed framework to show how service composition could be
automated. More specifically, we have demonstrated the applicability and evaluated the proposed approach using small well-known case studies. The evaluation step shows that a controller generated by our technique is able to prevent the system from violating the composition requirements.

Finally, it is worth mentioning that the use of the proposed framework is not limited to the service composition problem or the SOA domain. The SASCA framework is now a framework that can be used to model other systems where supervisory control theory based on FSMs may not be sufficient.

6.1 Summary

In this thesis, we have presented a survey on the problem of automated service composition, we found out that most approaches rely on a verification step to guarantee correctness of a composition. We also presented a literature review on DES and how it is increasingly being used to address some software engineering problems. Next, by noting that the contributions described in this thesis lie between service composition and DES with respect to the state-of-the-art; in Section 2.4, we presented the most closely related existing work to ours. In comparison to existing work, Section 2.4.1 reveals that no existing work has used DES to model the problem of service composition as done here.

In Chapter 3, we represented the core contributions of the thesis. We described how services can be represented using an existing industrial language and a formal language. We introduced the new service composition framework and various formalizations. To this end, we introduced a definition of controllability and the
notion of minimal restrictiveness. The intuition is that controllability is necessary and sufficient to solve the composition problem. The problem of automated service composition is formally presented and its solution was discussed. Next, we presented the new set of composition synthesis algorithms. The main results of this dissertation were presented in the forms of theorems and their proofs. These theorems guarantee the correctness of our approach. The correctness of the approach is further demonstrated by showing that the proposed algorithm terminates. Also, we provided a brief analysis of the computational complexity of the algorithms.

To back our theoretical claims, in Chapter 4, we presented a prototype implementation of the proposed technique. A system called SASCA for automated Web service composition was developed. We developed a translator that transforms a given WS-BPEL specification and its associated WSDL to an SLTS and back. We implemented the controller synthesis algorithm. Moreover, we discussed the compiler and the transformation rules in detail.

Finally, in Chapter 5, we provided an experimental evaluation of our approach. We employed well-known case studies in the area of service composition to demonstrate the applicability of our approach. The experiments we performed reveal that the generated controller prevents undesirable behaviours while otherwise restricting execution as little as possible. That is, the generated controller guarantees that the specification is always satisfied. In addition, we found out that the generated controller does not incur additional overhead when placed between the available services and the client. Furthermore, it is worth mentioning that the testing we performed using the Oracle WS-BPEL execution engine does not constitute comprehensive tests.
Thus, a lot can be done to improve the evaluation of the SASCA framework.

6.2 Limitations and Future Work

The framework presented in this dissertation relies on several assumptions in order to function. It will be interesting to relax some of these assumptions. For instance, we assumed a finite domain of variables and the transition system used in modeling the services are also finite. A formal language such as symbolic transition systems with more expressive power to model reactive systems and infinite domains could be adapted to model services in our framework. Another assumption we made is that the services we modeled are expected to communicate in a certain way by restricting the communication channel. The communication among services is assumed to be perfect in such a way that no single service can keep sending the same message infinitely often or wait for a specific message forever. In real world systems, such unexpected behaviours must be accounted for. The asynchronous communication treated in this dissertation does not rely on any specific implementation, however, when unbounded buffers are considered during message exchanges, the resulting state spaces may be infinite, and the problem becomes undecidable [28]. Thus, there is more room to further investigate the issue of compatibility checking (check unspecified reception and non-executable interaction) and the implementation of asynchrony.

In this framework, we assume full observability of events, but it will be of great interest to model partial observability aspects of services. That is, sometimes a service could progress from one state to another after executing some sequences of internal events or actions which cannot be observed by the controller. Hence, a new
control mechanism is needed in order to prevent the system from violating any system requirements. Partial observability could be modeled in our system by introducing $\tau$ as a special event into the SLTS definition.

One eminent extension of this work will be to investigate how non-functional requirements such as quality of services could be incorporated into the framework. A quality of service requirement is very important in the overall objective of service composition. To achieve quality of service in our framework, one can employ integer programming techniques [147] to determine the composition with the highest quality of services from different combination of services. In another direction, the issue of refinement of a composition requirement during failure solely relies on simulation relation, we envisage that a more efficient technique could be employed to assist the developer to refine his or her specification.

With respect to efficiency and effectiveness of compositions in our framework, there is a lot of work to do in the future. One such work is to look into how to efficiently represent the SLTSs for our formalism using data structures such as Binary Decision Diagrams or process algebra. We believe that more work could be done to improve and optimize the proposed algorithms. Another general extension of our approach could be the adapt modular or decentralized control of DES to model automated choreography synthesis of Web services. It is worth noting that the motivation behind a lot of service composition approaches suffer from becoming a reality due to the lack of existing benchmarks of Web services, or benchmark algorithms to provide a standard way of comparing one service composition technique with another. Thus, there was no way we could determine how well our approach
is doing with respect to other approaches during the evaluation process presented in Chapter 5. We had to rely on certain frequently used examples in the service domain. Therefore, a future work will be to create a uniform platform where our composition system could be compared with the other existing techniques. More specifically, it will be very interesting to compare our system in its full implementation and capabilities to some selected AI planning approaches. Furthermore, most of the examples and models used in our framework are limited to the e-business domain, but as far as the issue of correctness is concerned, our approach will be fully utilized if we consider safety critical systems and services in automotive, bioinformatics and the electric power domain.

In Chapter 4, we presented the implementation of the approach. However, the prototype developed needs to be improved in terms of automation and its user interface. In addition, the current implementation of the translator could be extended to include more advanced WS-BPEL constructs such as fault handing and scopes. In this direction, the notion of event enforcement can further be developed to consider WS-BPEL exceptions and fault handling constructs. Another future work would be to implement a module into the Oracle Business Process Management that can be used to monitor variables and the system at large at runtime that can work to aid event enforcement.

Finally, in this dissertation, we have shown how DES could be used to address the problem of service composition. However, there are many other research avenues where DES could be exploited to address certain problems in software engineering. One direct consequence of the motivation derived from this work would be to apply
DES or the framework presented in this thesis to the problem of API composition, Internet of things composition, and micro-services composition which are new in the field of software engineering. In addition, extending the framework to take into account service discovery, service selection and substitution will be of great interest.
Bibliography


[63] M.Y. Fayyad, A. Kamel, and A. Salah. ACUAI Framework for Automatic Composition of Web Services using Gaming AI. In Fifth International Conference on Digital Information and Communication Technology and its Applications (DICTAP), pages 1–6, Lebanese University, Lebanon, Apr 2015.


Appendix A

Algorithm 3.1 Java Source Codes for Chapter 4
Listing A.1: Algorithm 3.1

```java
/* Algorithm 3.1 Composer */
public Slts_Automaton controllerSynthesizer(Slts_Automaton plant, Slts_Automaton goal) {
  System.out.println("***************....INITIALIZING.......************");
  System.out.println("******....COMPUTING THE CONTROLLER..***************");
  Slts_Automaton controller = new Slts_Automaton();
  Slts_Automaton controllerK = new Slts_Automaton();

  // removeCommunicationErrors
  removeCommunicationErrors(plant);
  removeCommunicationErrors(goal);
  if (!simulationCheck(plant, goal)) {
    System.out.println("Spec cannot be simulated");
    Slts_Automaton refined = refineTargetToSimulatePlant(plant, goal);
    if (!(refined == null)) {
      goal = refined;
    } else {
      System.exit(1);
    }
  }

  System.out.println("spec simulated by plant proceeding...");

  int k = 1;
  Slts_Automaton Cnot = composition_Refinements(plant, goal);
  CreateDotFile.dumpSLTStoDot(Cnot, "Cnot.dot");
  // controllerK = Cnot.getCopystoSLTS();
  controller = Cnot.getCopystoSLTS();
  // controllerK = Cnot.getCopystoSLTS();

  do {
    // C k staticControllability(G W ) */
    controllerK = controller.getCopystoSLTS();
    controller = staticControllability(plant, Cnot);
    CreateDotFile.dumpSLTStoDot(controller, "StaticController" + k + ".dot");

    // unsafeStateMinimization */
    controller = unsafeStateMinimization(controller);
    CreateDotFile.dumpSLTStoDot(controller, "unSafeStateMiniController" + k + ".dot");

    // dynamicControllabilityAndGuardGeneration */
    controller = dyanamicControllablilityAndGuardGeneration(plant, controller);
    CreateDotFile.dumpSLTStoDot(controller, "dynamicController" + k + ".dot");
    System.out.println("No. of iterations=" + k);
    k++;
  }
}
```
while (!controllerK.equals(controller));
/* removeUnsafeState(C */
controller = removeUnsafeState(controller);
CreateDotFile.dumpSLTStoDot(controller, "unsafeStateRemoveController" + k + ".dot");
/* removeCommunicationErrors(C */
removeCommunicationErrors(controller);
/* reverseMessageDirection(C */
controller = reverseMessageDirection(controller);
CreateDotFile.dumpSLTStoDot(controller, "messageReverController" + k + ".dot");
controller.setName("Controller");
return controller;
}

Listing A.2: Code excerpt to initialize SASCA and execute it

public void testControllerSynthesizer() {
/* load first service */
Driver web1 = new Driver();
String loc = "data/sltsData/";
String fileName1 = "Plant1.txt";
String fileName = fileName1.substring(0, fileName1.indexOf(".txt")) + ".dot";
System.err.println("***** Parsing the SLTS from the file:");
System.out.println(loc + fileName1);
web1.loadSLTSAuto(loc + fileName1);
Slts_Automaton aut1 = web1.getWebService();
System.err.println("***** Displaying the Loaded SLTS*****");
LoggerFactory.getLogger(0).slts_Automaton_LogPrinter(aut1.getCopyOfSLTS());
dumpSLTStoDotFile(aut1, fileName);
/* load second service service */
System.err.println("Loading specifications now");
/* load Load spec service service */
Driver spec1 = new Driver();
// String fileSpec = "Spec1.txt";
String fileSpec = "Spec1.txt";
String fileSpec1 = fileSpec.substring(0, fileSpec.indexOf(".txt")) + ".dot";
System.err.println("***** Parsing the SLTS from the file:");
System.out.println(loc + fileSpec);
spec1.loadSLTSAuto(loc + fileSpec);
Slts_Automaton goal1 = spec1.getWebService();
System.err.println("***** Displaying the Loaded SLTS*****");
LoggerFactory.getLogger(0).slts_Automaton_LogPrinter(goal1.getCopyOfSLTS());
dumpSLTStoDotFile(goal1, fileSpec1);
/* initialize the controller synthesizer class */
Composition_Controller_Synthesizer syn = new Composition_Controller_Synthesizer();
/**
 * Find the Asynchronous parallel composition of all the
 */
ArrayList<Slts_Automaton> setofaut = new ArrayList<Slts_Automaton>();
setofaut.add(aut1);
// setofaut.add(aut12);
// Slts_Automaton assynProduct =
// syn.asynchronous_Parallel_Composition(setofaut);
/**
 * Testing Composer
 */
Slts_Automaton controller = syn.controllerSynthesizer(aut1, goal1);
RuntimeMonitoring runtimeInfor = syn.getRuntime();
LinkedHashMap<Inter_Transition, HashMap<Inter_Transition, Formula>> runDa = runtimeInfor.getRuntimeData();
// iterate over the map
for (Entry<Inter_Transition, HashMap<Inter_Transition, Formula>> entry : runDa.entrySet()) {
  // iterate over each entry
  Inter_Transition forcibleTransition = entry.getKey();
  HashMap<Inter_Transition, Formula> badTransAndillegalValues = runDa.get(forcibleTransition);
  for (Entry<Inter_Transition, Formula> en : badTransAndillegalValues.entrySet()) {
    Inter_Transition key = en.getKey();
    Formula value = en.getValue();
    System.err.println("Runtime Monitoring Information: ");
    System.out.println("forcible transition to enforce is");
    forcibleTransition.printTransInfor();
    System.out.println(" transitions that this forcible transition will prevent at runtime is ");
    key.printTransInfor();
    System.out.println(" when it takes the following values: " + value.toString() + "\n");
  }
}
System.err.println("<<<<<<<<<<<<<<<<< Controller Generation Successful>>>>>>>>>>>>>>");
LoggerFactory.getLogger(0).slts_Automaton_LogPrinter(controller);
dumpSLTStoDotFile(controller, "controller.dot");
Appendix B

Code Excerpts for Chapter 5

```xml
<?xml version = "1.0" encoding = "UTF-8" ?>
<!--
/////////////////////////////////////////////////////////////////////
Oracle JDeveloper BPEL Designer

Created: Sat May 27 11:05:15 EDT 2017
Author: lab
Type: BPEL 2.0 Process
Purpose: Asynchronous BPEL Process
/////////////////////////////////////////////////////////////////////
-->

<process name="Flight" targetNamespace="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
xmlns:client="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
xmlns:ora="http://schemas.oracle.com/xpath/extension"
xmlns:bpelx="http://schemas.oracle.com/bpel/extension"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xp20="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.functions.Xpath20"
xmlns:xref="http://www.oracle.com/XSL/Transform/java/oracle.tip.xref.functions.XRefXPathFunctions"
xmlns:hwf="http://xmlns.oracle.com/bpel/workflow/xpath"
xmlns:ids="http://xmlns.oracle.com/bpel/services/IdentityService/xpath"
xmlns:bpm="http://xmlns.oracle.com/bpel/workflow/xpath20/extensions"
xmlns:xref="http://www.oracle.com/XSL/Transform/java/oracle.tip.xref.functions.XRefXPathFunctions"
xmlns:ldap="http://schemas.oracle.com/xpath/extension/ldap"
xmlns:ui="http://xmlns.oracle.com/soa/designer"
xmlns:ns1="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight/correlationset">

<bpelx:annotation>
  <bpelx:property name="propertiesFile">
    <![CDATA[Wsdl/Flight_properties.wsdl]]>
  </bpelx:property>
</bpelx:annotation>

<bpelx:import namespace="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
location="Flight.wsdl" importType="http://schemas.xmlsoap.org/wsd1/"
ui:processWSDL="true"/>
```
<partnerLinks>
  <partnerLink name="flight_client" partnerLinkType="client:Flight"
    myRole="FlightProvider" partnerRole="FlightRequester"/>
</partnerLinks>

<variables>
  <!-- Reference to the message passed as input during initiation -->
  <!-- Reference to the message that will be sent back to the requester during callback -->
  <variable name="outputVariable" messageType="client:FlightResponseMessage"/>
  <variable name="callbackClient_FLOfferOP_InputVariable"
    messageType="client:FLOfferMsg"/>
  <variable name="isFlAvail" type="xsd:boolean"/>
    <from><literal>true</literal></from>
  <variable name="flAvailable" type="xsd:boolean">
    <from><literal>true</literal></from>
</variable>
  <variable name="Reply1_FLOfferOP_OutputVariable"
    messageType="client:FLOfferMsg"/>
  <variable name="Invoke1_FLOfferOP_InputVariable"
    messageType="client:FLOfferMsg"/>
  <variable name="Invoke2_FLNotAvailOp_InputVariable"
    messageType="client:FLNotAvailMsg"/>
  <variable name="OnMessage_FLConfirmOp_InputVariable"
    messageType="client:FLconfirmMsg"/>
  <variable name="OnMessage_FLRejectedOp_InputVariable"
    messageType="client:FLRejectedMsg"/>
</variables>

<correlationSets><correlationSet name="FlightCor" properties="ns1:Property1"/></correlationSets>

<sequence name="main">
  <!-- Receive input from requestor. (Note: This maps to operation defined in Flight.wsdl) -->
  <receive name="fRequest" partnerLink="flight_client" portType="client:Flight"
    operation="FLRequestOp" variable="FlData" createInstance="yes">
    <correlations>
      <correlation set="FlightCor" initiate="yes"/>
    </correlations>
  </receive>

  <assign name="checkFl_avail">
    <copy>
      <from>$flAvailable</from>
      <to>$isFlAvail</to>
    </copy>
  </assign>

  <if name="If_Avalablity">
    <documentation>
      <!\[CDATA[flightAvail]\]>
    </documentation>
    <condition>$isFlAvail</condition>
    <sequence name="Sequence3">
      <sequence name="Sequence1">
        <assign name="processBooking">
          <copy>
            <from>$FlData.flightInfor/client:FlightType</from>
            <to>$callbackClient_FLOfferOP_InputVariable.payload/client:result</to>
          </copy>
          <copy>
            <from>$FlData.flightInfor/client:FlightType</from>
            <to>$Invoke1_FLOfferOP_InputVariable.payload/client:result</to>
          </copy>
        </assign>
        <invoke name="Invoke1" partnerLink="flight_client" portType="client:FlightCallback"
          operation="FLOfferOP" inputVariable="Invoke1_FLOfferOP_InputVariable"
          bpelx:invokeAsDetail="no">
          <correlations>
            <correlation set="FlightCor"/>
          </correlations>
        </invoke>
      </sequence>
    </sequence>
  </if>
</sequence>

```xml
<pick name="Pick1">
  <onMessage partnerLink="flight_client" variable="OnMessage_FLConfirmOp_InputVariable" portType="client:Flight" operation="FLConfirmOp">
    <correlations>
      <correlation set="FlightCor"/>
    </correlations>
    <empty name="Empty1"/>
  </onMessage>
  <onMessage partnerLink="flight_client" variable="OnMessage_FLRejectedOp_InputVariable" portType="client:Flight" operation="FLRejectedOp">
    <correlations>
      <correlation set="FlightCor"/>
    </correlations>
    <exit name="Exit1"/>
  </onMessage>
</pick>

<sequence name="Sequence2">
  <assign name="processNotAvail">
    <copy>
      <from>$FlData.flightInfor/client:Dates</from>
      <to>$callbackClient_FLOfferOP_InputVariable.payload/client:result</to>
    </copy>
    <copy>
      <from>$FlData.flightInfor/client:FlightType</from>
      <to>$Invoke2_FLNotAvailOp_InputVariable.payload/client:result</to>
    </copy>
  </assign>
  <invoke name="Invoke2" partnerLink="flight_client" portType="client:FlightCallback" operation="FLNotAvailOp" inputVariable="Invoke2_FLNotAvailOp_InputVariable" bpelx:invokeAsDetail="no">
    <correlations>
      <correlation set="FlightCor"/>
    </correlations>
  </invoke>
</sequence>

<else>
  <documentation><![CDATA[flightNotAvail]]></documentation>
  <sequence name="Sequence2">
    <assign name="processNotAvail">
      <copy>
        <from>$FlData.flightInfor/client:Dates</from>
        <to>$callbackClient_FLOfferOP_InputVariable.payload/client:result</to>
      </copy>
      <copy>
        <from>$FlData.flightInfor/client:FlightType</from>
        <to>$Invoke2_FLNotAvailOp_InputVariable.payload/client:result</to>
      </copy>
    </assign>
    <invoke name="Invoke2" partnerLink="flight_client" portType="client:FlightCallback" operation="FLNotAvailOp" inputVariable="Invoke2_FLNotAvailOp_InputVariable" bpelx:invokeAsDetail="no">
      <correlations>
        <correlation set="FlightCor"/>
      </correlations>
    </invoke>
  </sequence>
</else>

<if>
  <invoke name="callbackClient" partnerLink="flight_client" portType="client:FlightCallback" operation="FLOfferOP" inputVariable="callbackClient_FLOfferOP_InputVariable" bpelx:invokeAsDetail="no"/>
</if>
</process>
```
Listing B.2 WSDL Code Excerpt for Flight Service

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<wsdl:definitions
    name="Flight"
    targetNamespace="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:client="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:cor="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight/correlationset"
    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
    xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
>
    <plnk:partnerLinkType name="Flight">
        <plnk:role name="FlightProvider" portType="client:Flight"/>
        <plnk:role name="FlightRequester" portType="client:FlightCallback"/>
    </plnk:partnerLinkType>

    <vprop:propertyAlias propertyName="cor:Property1" messageType="client:FLRequestMsg" part="cor"/>
    <vprop:propertyAlias propertyName="cor:Property1" messageType="client:FLofferMsg" part="cor"/>
    <vprop:propertyAlias propertyName="cor:Property1" messageType="client:FLRejectedMsg" part="cor"/>
    <vprop:propertyAlias propertyName="cor:Property1" messageType="client:FLconfirmMsg" part="cor"/>
    <vprop:propertyAlias propertyName="cor:Property1" messageType="client:FLNotAvailMsg" part="cor"/>

    <wsdl:import namespace="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight/correlationset"
        location="WSDLs/Flight_properties.wsdl"/>
    <wsdl:types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema">
            <import namespace="http://xmlns.oracle.com/TravelCaseStudy/TravelExample/Flight"
                schemaLocation="xsd/Flight.xsd"/>
        </schema>
    </wsdl:types>
    <wsdl:message name="FLRequestMsg">
        <wsdl:part name="flightInfor" element="client:FlightprocessRequest"/>
        <wsdl:part name="cor" element="client:string"/>
    </wsdl:message>
    <wsdl:message name="FLconfirmMsg">
        <wsdl:part name="payload" element="client:process"/>
        <wsdl:part name="cor" element="client:string"/>
    </wsdl:message>
    <wsdl:message name="FLRejectedMsg">
        <wsdl:part name="payload" element="client:process"/>
        <wsdl:part name="cor" element="client:string"/>
    </wsdl:message>
    <wsdl:message name="FlightResponseMessage">
        <wsdl:part name="payload" element="client:processResponse"/>
        <wsdl:part name="cor" element="client:string"/>
    </wsdl:message>
</wsdl:definitions>
```

```xml
<wSDL:message name="FLNotAvailMsg">
    <wSDL:part name="payload" element="client:processResponse"/>
    <wSDL:part name="cor" element="client:string"/>
</wSDL:message>
<wSDL:message name="FLOfferMsg">
    <wSDL:part name="payload" element="client:processResponse"/>
    <wSDL:part name="cor" element="client:string"/>
</wSDL:message>
<wSDL:portType name="Flight">
    <wSDL:operation name="FLRequestOp">
        <wSDL:input message="client:FLRequestMsg"/>
    </wSDL:operation>
    <wSDL:operation name="FLConfirmOp">
        <wSDL:input message="client:FLconfirmMsg"/>
    </wSDL:operation>
    <wSDL:operation name="FLRejectedOp">
        <wSDL:input message="client:FLRejectedMsg"/>
    </wSDL:operation>
</wSDL:portType>
<wSDL:portType name="FlightCallback">
    <wSDL:operation name="processResponseOp">
        <wSDL:input message="client:FlightResponseMessage"/>
    </wSDL:operation>
    <wSDL:operation name="FLNotAvailOp">
        <wSDL:input message="client:FLNotAvailMsg"/>
    </wSDL:operation>
    <wSDL:operation name="FLOfferOP">
        <wSDL:input message="client:FLOfferMsg"/>
    </wSDL:operation>
</wSDL:portType>
</wSDL:definitions>
```