Iowa State University
Digital Repository

Graduate Theses and Dissertations | Iowa State University Capstones, Theses and Dissertations

2008

An iterative approach towards web service composition using feedback from analysis of composition failures

Dinanath Nadkarni
Iowa State University

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

Part of the Computer Sciences Commons

Recommended Citation

Nadkarni, Dinanath, "An iterative approach towards web service composition using feedback from analysis of composition failures" (2008). *Graduate Theses and Dissertations*. 10896.
https://lib.dr.iastate.edu/etd/10896

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

An iterative approach towards web service composition using feedback from analysis of composition failures

by

Dinanath Nadkarni

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

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

Iowa State University

Ames, Iowa

2008

Copyright © Dinanath Nadkarni, 2008. All rights reserved.

TABLE OF CONTENTS

LIST OF FIGURES iv

ACKNOWLEDGEMENTS vi

ABSTRACT vii

CHAPTER 1. INTRODUCTION 1

1.1 Thesis Outline 3

CHAPTER 2. BACKGROUND 4

CHAPTER 3. WEB SERVICE COMPOSITION 9

3.1 Introduction 9

3.2 The Composition Algorithm 13

CHAPTER 4. FAILURE ANALYSIS AND RECOVERY 17

4.1 Goal of Failure Analysis 17

4.2 Approach towards Failure Analysis 18

CHAPTER 5. CASE STUDIES 24

5.1 Library Book Reservation System 24

5.2 Stadium Reservation System 30

CHAPTER 6. CORRECTNESS AND COMPLEXITY ANALYSIS 34

6.1 Correctness 35

6.2 Complexity Analysis 38

CHAPTER 7. IMPLEMENTATION 42

7.1 Service Creation and Representation 43

7.2 Architecture 44
7.2.1 Web Service Representation 44
7.2.2 Processing Module 44
7.3 Demonstration 46
7.3.1 Demonstration: E-Library case study 46
7.3.2 Demonstration: Stadium Reservation System Case Study 50
CHAPTER 8. CONCLUSION AND FUTURE WORK 53
8.1 Contribution and Future Work 53
8.2 Conclusion 55
APPENDIX A. COMPOSITION ALGORITHM 57
APPENDIX B. FAILURE ANALYSIS 60
BIBLIOGRAPHY 66

LIST OF FIGURES

Figure 2.1 Two different types of composition [2]: (a) Orchestration based (b) P2P based 5
Figure 2.2 Goal service specified at an abstract level 6
Figure 3.1 Web service as a Symbolic Transition System 10
Figure 4.1 Computation tree in a failure scenario 19
Figure 5.1 e-Library: initial goal service 25
Figure 5.2 e-Library: Component Services 26
Figure 5.3 e-Library: suggested modification to the goal service after first failure . 27
Figure 5.4 e-Library: final suggested goal service 28
Figure 5.5 e-Library: choreographer corresponding to the final suggested goal service 29
Figure 5.6 Stadium Reservation: Initial goal service 31
Figure 5.7 Stadium Reservation: Component services 32
Figure 5.8 Stadium Reservation: Suggested goal service with failed transition . . 33
Figure 5.9 Stadium Reservation: Partially generated choreographer 33
Figure 7.1 MOSCOE Service Composition tool: screenshot 42
Figure 7.2 MOSCOE: Creating a guard on a transition 43
Figure 7.3 UML diagram depicting the web service data 44
Figure 7.4 UML diagram depicting the composition manager classes 45
Figure 7.5 The e-Library goal service in MoSCoE 46
Figure 7.6 e-Library component services and goal service selected 47

Figure 7.7 e-Library status message 48
Figure 7.8 e-Library final suggested goal service 48
Figure 7.9 e-Library choreographer corresponding to the final suggested goal service 49
Figure 7.10 MoSCoE: Stadium Reservation goal service 50
Figure 7.11 MoSCoE: Stadium Reservation composition status message 51
Figure 7.12 Stadium Reservation System: final suggested goal service 51
Figure 7.13 Stadium Reservation System: Choreographer corresponding to suggested goal service 52

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to all the people who helped with the various aspects of my research and the writing of this thesis. First and foremost, I would like to thank my major professor Dr. Robyn Lutz for the guidance and support that she provided for the entire span of time that I was in graduate school. I greatly appreciate her patience and advice in matters concerning this research as well as thesis writing.

I would also like to thank Dr. Samik Basu and Dr. Vasant Honavar for their willingness to serve on my POS committee. A special thanks to Dr. Samik Basu for his insight and guidance into my research and for taking the time to help me with all the questions that I had during my research.

I would like to thank Dr. Jyotishman Pathak for his contribution towards MoSCoE, on which this work is based. I would also like to thank Mohammed Alabsi, Melissa Yahya, Rakesh Setty and Mahantesh Hosamani for their contribution towards the implementation of the MoSCoE composition algorithm. In addition, I would also like to thank Linda Dutton for her guidance and assistance in all matters related to graduate school.

Last but not the least, I would like to thank my parents Laxmicant and Revati and my brother Mukul for their unwavering support and guidance.

ABSTRACT

The Web service composition problem involves the creation of a choreographer that provides the interaction between a set of component services, to realize a goal service. It is desirable to have an automated or semi-automated composition process that accepts a business requirement specification in the form of a goal web service and automatically creates a composite service from a set of available component web services.

A one-step composition process is inadequate in scenarios where the complete requirements are not known or the user is unaware of the functionality provided by the component services. In such cases, composition of the goal web service might fail, because of an incomplete specification of the goal service or because the specified goal service cannot be composed using the available component services. An iterative approach towards web service composition would help to address failures that can occur during composition, by providing the user with feedback as to the cause of the failure and possible recovery solutions to the failure. The user can then change the goal service based on this feedback to arrive at a successful composition.

This work addresses the analysis of failures that occur during composition and the type of feedback to be provided to the user. The approach first identifies the cause of failure and explores all possible recovery options for the failure. If possible, suggestions are made to modify the goal service based on the solution provided by every recovery option. The composition process is then simulated on the modified goal services to detect future failures. From amongst these modified goal services, the service which has the least failures is then provided to the user as feedback.

The main contribution of this work is an approach towards analysis and recovery from failures that occur during web service composition. The web service composition algorithm is

explained, the goal of failure analysis and recovery is described and the approach towards failure analysis is then described in detail. This is followed by a demonstration of the implementation of the algorithm in two case studies.

CHAPTER 1. INTRODUCTION

One benefit of Web Service Composition is that it allows a web service or an application developer to derive a very specific functionality from a set of independently developed components, in an economical and time saving manner. This concept provides a high potential for reusability and also allows the developer of the application to focus on the business logic, instead of the technical intricacies. In this context, the developer is a person involved in the creation or development of the web service or the business application. The client is defined as the entity that uses the developed application or web service. The client can be a person or another application or web service.

Many approaches towards automated or semi-automated web service composition have been investigated, using concepts of AI planning and workflow composition [9]. One such approach is based on the translation of the BPEL4WS specification into state transitions systems[10] and the specification of the goal service in a formal requirements language known as EAGLE[11]. Another approach denotes the composite service in terms of a process model[12]. The component services are selected during the execution of the composition and not during the design stage. The selection of component services is formulated as an optimization problem and solved using linear programming methods. Carman et al. [13] treat the web service composition as a planning problem and use a combination of a semantic type matching algorithm and an interleaved search and execution algorithm to automate the web service composition.

A single step composition approach suffers from one drawback; if the required goal service cannot be realized in the first step, the composition process fails. The developer then has to create a new goal service possibly without having any idea as to why the composition failed. Furthermore, it forces the developer to specify a complete goal service on the first go,

which is not always possible. It would be highly desirable to allow the developer to specify an incomplete goal service and build on it based on feedback provided by the composition process. In such cases, if any failures occur, the developer would be able to reformulate the goal service based on the feedback given. This feedback based, iterative composition process is followed in MoSCoE [3]. The developer can start off with an abstract or incomplete goal service, and reformulate the goal service in subsequent iterations based on feedback received from the composition process.

MoSCoE is inspired by the COLOMBO framework presented by Berardi et al. [6]. This framework describes the behavior of web services using finite state transition systems. In MoSCoE, Web services are represented as Symbolic Transition Systems [3]. The developer specifies a business requirement in terms of a Symbolic Transition System known as the goal service. The developer then selects a set of services which can be used to compose the goal service. Such services are known as the component services. A set of component services from which a subset of component services can be selected to take part in composition is known as the services repository. Component services are also expressed as Symbolic Transition Systems.

The web service composition algorithm creates a choreographer that acts as a communication interface between the goal service and the component services, in a manner so as to realize the goal service. The choreographer is therefore, a composite service that interacts between the goal service and the component services via message passing, to realize the functionality of the goal service. The composition process is described in more detail in chapter 3.

An iterative composition process must provide the user with feedback in the event of composition failure. This feedback should provide the developer with a description of the cause of failure, as well as a possible recovery from the failure. As a side effect, the feedback might also provide the developer with a fair idea of the functionality of the component services. This knowledge would help the developer to modify the goal service accordingly, so as to avoid the failure in the subsequent iterations.

This work describes one such approach of providing feedback to the user. When failure occurs, the cause of the failure is identified and possible recovery options from this failure are

explored. Duplicate goal services are created and each duplicate service is modified according to each of the possible recovery options. The composition process is then simulated on these modified goal services to ensure that failures do not occur in these modified services in the future. One of the modified goal services and the choreographer corresponding to that goal service are then presented to the developer as feedback. This approach is described in detail in chapter 4.

This work then describes the composition and failure recovery process in two case studies, the Library book reservation system and the Stadium reservation system.

1.1 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2: This chapter provides background information on web services and iterative web service composition.

Chapter 3: This chapter provides an introduction to web service composition. The representation of services is described and the composition algorithm is described in considerable detail.

Chapter 4: The Failure analysis and recovery approach is described in complete detail in this chapter.

Chapter 5: Two case studies are described in this chapter. The case studies illustrate the composition process and the failure analysis and recovery for a Library book reservation system and a Stadium reservation system.

Chapter 6: This chapter describes the correctness of the composition algorithm. The complexity analysis of the composition algorithm and the failure analysis is then described.

Chapter 7: MoSCoE provides an open source composition tool which implements the composition process described in this work. This chapter demonstrates the functioning of this tool for the service composition in the case studies.

Chapter 8: The conclusion of this thesis is presented here. Some research directions for future work are also listed.

CHAPTER 2. BACKGROUND

A Web service is a web application which is available on the web via a URL and which can be found and accessed using standard XML based protocols, such as SOAP and WSDL. A web service can be considered as a reusable component that can be used to fulfill part of the business requirement for an application. Formally, 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 SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [1].

Based on the design, availability and functionality of web services, it is possible to realize a business requirement by combining the functionality of different web services, either by accessing the web services in parallel or in a sequence. This process is commonly known as web service composition. There are two different types of web service compositions: orchestration based and P2P based [2].

In orchestration based composition, the services interact by exchanging messages amongst each other through a common endpoint, known as the mediator [6]. The mediator manages the interaction among the component services by controlling the exchange of messages between them. P2P based compositions do not use a choreographer, rather the component services interact directly with the client to arrive at a composite goal service. The two approaches towards service composition have been illustrated in Figure 2.1. This work uses an orchestration based algorithm to compose web services.

In many cases, the complete requirements for an application are not available at the be-

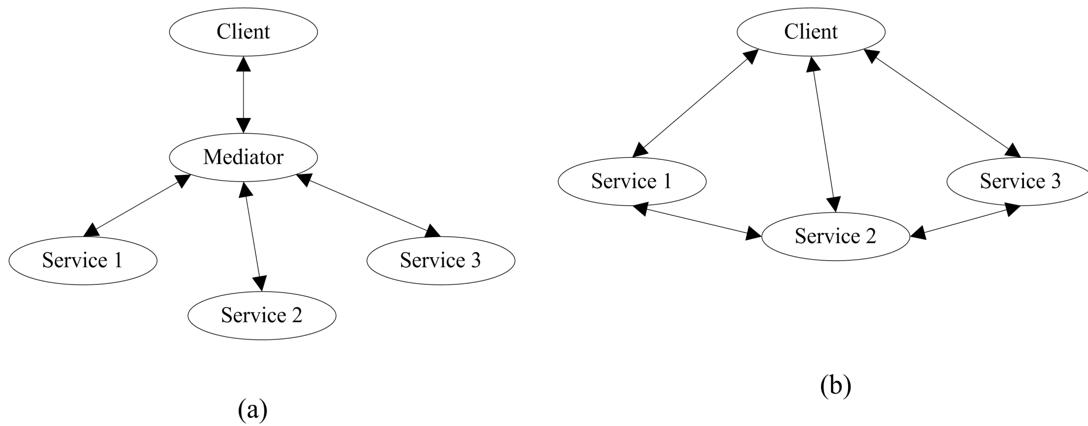

Figure 2.1 Two different types of composition [2]: (a) Orchestration based (b) P2P based

ginning. Also, new requirements may be introduced during the later iterations of software development. A web service developer tasked with creating a goal service based on such requirements will therefore specify an incomplete service, with the intention of building up this service as new requirements arrive. Furthermore, since the components used in a composition are often fetched from disparate and independent sources, it might not be possible for a developer to specify the goal service exactly in terms of the functionality provided by the component services. The developer will therefore, specify the goal service at an abstract level and refine the goal based on how the composition process goes.

We illustrate the specification of an abstract and possibly incomplete goal specification with an example. Consider a scenario where a developer is asked to create a shopping cart processing system from a set of component services. The developer is also told that the current goal service is required only to process items in the shopping cart and charge the customer. The system would later be extended with a delivery processing feature. The developer would start out by creating a goal service with only the functionality required to process shopping cart items. If the developer is not completely aware of the details involved in processing of shopping cart items, he would specify the goal service at an abstract level. For instance, the entire processing of shopping cart items might be abstracted using a single function processShoppingCart(). Similarly, the credit card processing function can be abstracted by specifying a single function

chargeCreditCard(). This shopping cart goal service has been illustrated in Figure 2.2. As shown, the processShoppingCart() and chargeCreditCard() functions have been abstracted and the goal service does not contain any functionality for delivery processing.

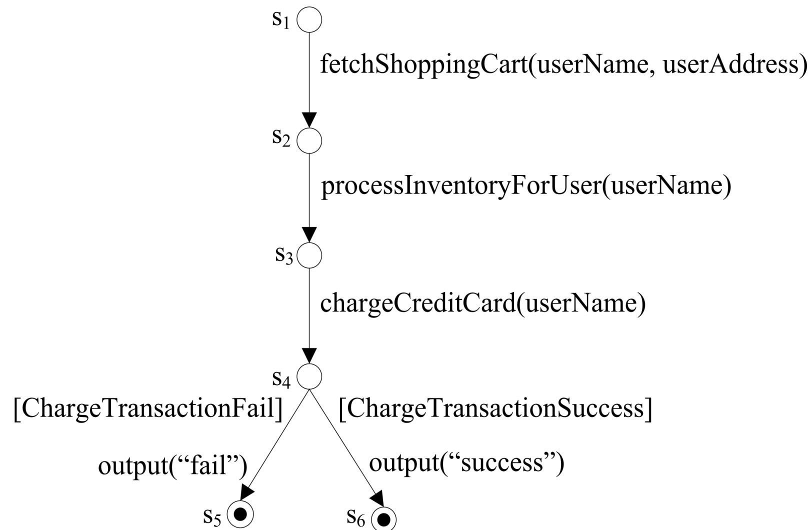

Figure 2.2 Goal service specified at an abstract level

A single step composition process will not work in such circumstances. In the shopping cart scenario, the initial goal service specified by the developer would fail to compose if the component services did not have the functions required for it. Consider for example, that the developer has the following functions available in the component services: findItemsInInventory(), obtainDeliveryDetailsByCustomerId(), reserveItemsForDelivery(), chargeVisaCard(), chargeDiscoverCard() and chargeMasterCard(). Since none of these functions are mentioned in the goal service, composition of the goal service using these components will fail. In such scenarios, an iterative approach towards composition is needed [3]. The developer is given feedback at the end of each iteration of the composition process, allowing the developer to refine the goal service based on such feedback in subsequent iterations. This work is based on such an iterative composition process [3].

An iterative approach for composition requires that the service developer be given feedback during the composition process. If the goal service can be composed in a single iteration, then the only feedback required would be the composite service. In some cases however, it might

not be possible to compose the goal service in a single iteration. In such cases, the composition process could fail to create a composite service in some iterations. The cause of this failure must be identified in order to formulate an appropriate feedback message for the user. It would be advantageous for the composition process to automatically attempt to recover from a failure. This would have two advantages; it would be able to determine how the user can change the goal service to avoid the failure and in doing so, it would provide the user with some information about the functionality provided by the component services. This work focuses on the analysis of failures and approaches towards recovery from failures that occur during composition.

When composition fails during an iteration, the algorithm identifies the cause of the failure and automatically attempts to modify the goal service in a manner so as to avoid this failure. The user is provided with a list that describes the failures that occur during the iteration, along with the modified goal service and the composite service corresponding to the modified goal service. This form of feedback has a number of advantages:

1. The list of failures contains descriptions about the cause of each failure. This will provide the user some information about the functionalities and requirements of the component services. For example, if a failure occurred due to unavailability of some input data, the detailed message would indicate the component that requires the data.

2. A goal service could be modified in different ways to recover from a failure, based on the component services being used in the composition. This approach would explore all such modified services and provide the best composite service to the user, based on user specified criteria. The advantage here is that all the possible modifications to the goal service are considered in an iteration, saving the user from trying out different modifications to the goal service manually.

3. If the user decides to keep the modified goal service, the composition process need not be repeated since the composite service corresponding to the modified goal service is already provided along with the modified goal service.

A detailed explanation on the goals of failure analysis and recovery and the approach taken towards failure recovery is presented in chapter 4.

CHAPTER 3. WEB SERVICE COMPOSITION

3.1 Introduction

Web services are normally described using Web Service Definition Language (WSDL) [8]. In this work, web services are described in terms of Symbolic Transition Systems (STS). An STS is a transition system with guards on transitions and state variables on an infinite domain [2]. An STS provides an intuitive way of visualizing a web service as a transition diagram, with each state representing a specific state of the web service process and transitions leading the service from one state to another. A web service represented by an STS consists mainly of three types of transitions: a transition that accepts input from the client, a transition that performs an atomic action or function and a transition that provides output to the client. In STSs, these three types of transitions are represented by input, output and atomic actions respectively. The three actions are described below:

- Input action indicates the receipt of data. Such actions indicate that the client has passed a message to a component or service. Input actions have the format ?msgHeader(msgSet). The ? at the start of the action indicates that the action is an input action. msgHeader is the name of the input action, and msgSet is the message content that is passed via the input action. For example, ?studentDetail(ID, Address) is an input action which inputs variables ID and Address into the system.

- Output actions indicate the sending of data. They are used to indicate the sending of a message to a client or to another web service. An output action is denoted by !msgHeader(msgSet), where msgHeader is the name of the output action and msgSet is the content of the message being passed from the component or goal service. For

example, !studentGrade(sGrade) is an output action named studentGrade that passed sGrade as the message.

- In terms of STSs, atomic actions represent function calls. An atomic action is of the form funcName(I; O) where set I represents the input message or input arguments to the function and O is the output value returned by the function. For example, atomic action add(x,y; z) represents a function that adds two integers x and y and returns the sum in variable z.

Every transition can have a guard condition associated with it. A guard condition must be satisfied if that transition is to be carried out. In the STS representation of the goal service, guard conditions are enclosed in square brackets.

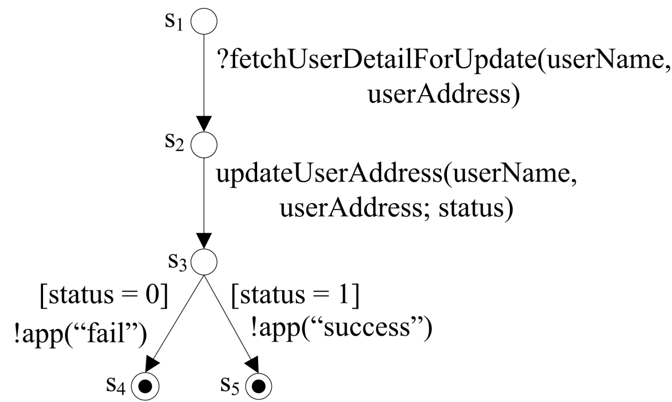

Figure 3.1 Web service as a Symbolic Transition System

Figure 3.1 illustrates a sample web service expressed as an STS. The action *?fetchUserDetailForUpdate(userName, userAddress)* indicates the receipt of a message with variables *userName* and *userAddress* by the service. The client passes these variables on to the goal service. The Atomic action *updateUserAddress* takes in *userName* and *userAddress* and returns a status indicating whether the *address* update was successful or not. The status is checked at the guard condition on the outgoing transitions on state s_3 . If the update was successful (status = 1), the service outputs "success" via the output action *!app("success")*, otherwise it outputs a "fail" message. Guard conditions exist on the outgoing transitions from state s_3. Dependencies among component stores are specified in terms of flow links [3].

The goal service T_g and the component services $T_1, \ldots, T_n$ are specified as STSs. The composition process attempts to identify a subset of the component services that when composed with a choreographer STS T_{cr} realizes the goal service T_g. The choreographer orchestrates this by passing messages to and from the component services and the goal service. The choreographer adopts the input and output actions of the goal service and realizes the goal service by making the appropriate calls to atomic actions provided by the component services. Thus the choreographer acts as an interface that interacts with the component services and the goal service, in a manner so as to replicate the functionality of the goal service using the component services. The choreographer is also responsible for storing data that is passed from the component services or from the goal service, so that the data might be used at the appropriate time.

Communication between the component services and the choreographer takes place via message passing. If a component service houses a function call that takes parameters, it must take the parameters to the function call through an input action. To call this function, the choreographer sends a message to the component service via an output action. This message is received by the component service via an input action specified in its STS. The component service can then execute its function. The component service uses an output action to pass the results of the function call back to the choreographer, which indicates the receipt of this message through an input action.

The composition algorithm described in this work is based on the composition algorithm used in MoSCoE [3]. The composition algorithm has been modified so as to facilitate the failure analysis and recovery process. These changes are described below, followed by a description of the composition algorithm. The changed composition algorithm is shown in Appendix A.

(a) **Inclusion of individual message stores for each of the component services**. The original composition algorithm [3] maintained a message store only for the choreographer. The algorithm used in this work maintains message stores for the individual components in addition to the choreographer store. The message store for a component is used to hold the messages received by that component as well as the messages that are encountered

when that component is traversed. If the goal service has a transition on an atomic action and the choreographer store does not have the required input message set, but the message set is present in some component store, it indicates that the component has to be traversed prior to processing the current transition in the goal service. Having such component message stores allows the components to store messages, so the choreographer can ask for the message only when required while composing the goal service. This reduces the creation of unused transitions in the choreographer. For example, if a function call on a component service returns an output message which is not used in the goal service, the choreographer need not send a message to the component service asking for it. If the goal service needs this message later on, the choreographer will then request the message from the component service.

(b) Handling of the guard conditions. If a guard condition requires variables that are not present in the choreographer store, the component stores are checked to see if any of the components have encountered those guard variables. If some component does have those variables, it is traversed so as to make the guard variables available to the choreographer. This differs from the original composition algorithm which would mark the missing guard variables as a failure.

(c) Handling of missing parameters in function calls. If a function invocation cannot be processed because the choreographer cannot provide input parameters to the function, the component stores are searched for those missing parameters. If any component store has the required parameters, it indicates that the component must be parsed as well, so as to provide the choreographer with the missing parameters. In comparison, the original algorithm would mark missing parameters as a failure.

(d) Handling of atomic actions. If an atomic action is encountered in a component service, a transition on the atomic action is not inserted into the choreographer immediately. The message store of the component service is updated with the variables of the atomic action and the atomic action is marked as done. If the atomic action in the component

has an output value, this output value is retained in the component store. This way, if the output value of the atomic action is required in the choreographer, the component service can be traversed again to find an output action which sends the output message to the choreographer. If the atomic action does not have any output value or the output value is not used in the goal service, the choreographer can avoid creating a transition on it.

3.2 The Composition Algorithm

The composition algorithm parses the goal service one state at a time and creates the choreographer T_{cr} based on the kind of actions and guard conditions encountered at every state of the goal service. The choreographer maintains a repository of the messages received during the composition process. The individual components also maintain their message stores.

The entry point to the composition algorithm is the function $GENERATE(r, [s_1,\ldots,s_n], [R_1,\ldots,R_n], t, G, R)$. Here, r is the start state of the goal STS, $s_1, \ldots, s_n$ are the start states of the component services and $R_1, \ldots, R_n$ are the variable repositories for each component. Each component repository R_i holds the input and output messages encountered while interacting with component T_i. t is the current state of the choreographer. The guard condition G is initialized to true and R holds the input and output message headers encountered by the choreographer. A global set done keeps track of the function invocations in the composition.

There are 4 cases to consider:

Case 1: Input from the client. A transition with an input action in the goal service indicates input from the client. When an input action is encountered in the goal service, a corresponding transition is created in the choreographer. Choreographer store R is then updated with the variables used in the action. This indicates that the choreographer now has the input message and can use it anytime in the course of the service composition. At this point, the message stores of the component repositories are not updated. This case corresponds to the receipt of a message from the client.

Case 2: Output to the client. If there is a transition on an output action in the goal

service, a corresponding transition is created in the choreographer if the message for the output action is available (present in choreographer store R). If the choreographer does not hold the message set, it might be possible that the output message is present in the message store of a component service. Hence the component stores are searched and if the message is found in any component store, the algorithm attempts to find an output action in that component which will provide the message to the choreographer. This case corresponds to the passing of a message to the client. If the output message is not available, a failure is reported.

Case 3: Function invocation. The processing of a function invocation or atomic action in the goal service is dependent on the availability of the input message for that function and on the current states of the service components providing that function. There are three sub-cases:

(a) **The input message required for the function invocation is not in the choreographer store R but is present in the message stores of the component services**. Variables are inserted into the message store of a component service when they are encountered while traversing the component service or when the choreographer sends a message to the component. If the variables are present in a component message store, an output action is required to send those variables to the choreographer. The composition algorithm searches for such an output action in the corresponding component service. If it finds the output action, a corresponding transition is created in the choreographer, the current states of the choreographer and the component service are advanced and the composition method is recursively invoked. For example, if the function in the goal service is $fname(I;O)$, $I = i_1, i_2$ and both i_1 and i_2 are not available in the choreographer store, the algorithm searches for components which have the variables i_1 and i_2 in their message stores. If such component services are found, the algorithm attempts to find output action on i_1 and i_2 in those components so that variables i_1 and i_2 can be passed to the choreographer.

(b) **Current transition in the goal service has a function invocation and none of the component services provide a transition on that action from their current states**. The algorithm selects a component T_i that provides the required function. There are four sub-cases:

(i) The current state s_i of component T_i is an input action for which the choreographer can provide the input message

(ii) s_i is an input action for which the choreographer cannot provide the input message

(iii) s_i has an output action

(iv) s_i has a transition on a function invocation

In cases (i) and (iii), corresponding transitions on input or output actions are created in the choreographer, variables are inserted into the message store of the component and R, followed by a recursive call to procedure $GENERATE$.

In case (ii), flow links [3] are explored in an attempt to find an output action that will provide the message set for the input action. If no such flow links exist, failure recovery action is initiated. If there is a component T_j that can provide the output action, the current state of component T_j is examined and the set of transitions leading to the required output action are followed. If the current transition on component T_j is an output action or an input action for which the choreographer can provide the message set, the corresponding transition is created in the choreographer and store R is updated followed by a recursive call to $GENERATE$.

Case (iv) applies if T_j has a transition on a function call and the choreographer store can provide the input message to this function. In this situation, the function is marked as done and $GENERATE$ is recursively called. If the input message at s_j cannot be provided, flow links are followed again in similar manner.

(c) **The transition in the goal service has a function invocation and some component provides a transition on that action from its current state**. In this case, the behavior of the algorithm depends on the availability of the input

message. If the choreographer can provide the input message for this function, the function is marked as done, the message is inserted into store R as well as the corresponding component store and procedure $GENERATE$ is recursively called. If the function has an output value, a transition on the output message for that valuation is not created in the choreographer at this point. Instead the transition is created when it is called for in the goal service.

(d) Variables in the guard are not available in the choreographer store R. If all of the guard variables are available in the component stores, those components are explored one by one to find output actions which will pass the required variables to the choreographer. This results in the creation of a series of transitions in the choreographer so as to make the guard variables available to the choreographer.

The composition algorithm is shown in Appendix A.

CHAPTER 4. FAILURE ANALYSIS AND RECOVERY

Service composition can fail for a number of reasons, some of which have been mentioned in chapter 3. This chapter describes the goal of failure analysis, lists the reasons for composition failures and describes the actions taken to recover from failures.

4.1 Goal of Failure Analysis

As mentioned earlier, the feedback given to a user is of utmost importance in iterative service composition. If a composite service can be devised in a single iteration without any failure, the user can simply be given the composite service itself. However, if the composition process fails during an iteration, the user must be given feedback as to the cause of the failure and how to resolve the failure. This raises the question of what is to be given as feedback and how that feedback is to be presented to the user.

When failure occurs in composition, the first step towards failure analysis is to identify the source of the failure. If the goal service can be changed so that the failure can be avoided, it would help the user if the changes required to resolve the failure were presented. If it is not possible to recover from the failure at all, a partial list of changes would still help the user acquire a better understanding of the functionality provided by the component services.

The feedback mechanism described in this work attempts to make suggestions that modify the goal service so as to avoid the failure. If recovery from the failure is not possible, the goal service is modified as much as possible in an attempt to avoid the failure and a partially modified goal service is presented to the user. This approach provides both of the requirements of feedback described in the previous paragraph.

4.2 Approach towards Failure Analysis

When failure occurs during composition, the cause of the failure is identified. Failures occur mostly due to missing message sets or missing function invocations. Most failures are resolved by finding alternate paths in the component services or by calling on components that provide the missing message sets. Hence, multiple resolutions are possible for most failures. The failure analysis approach presented in this work explores each such recovery option. On failure, we identify all possible recovery options. For every possible recovery solution, a new branch of computation is created with its own copy of the component services, the goal service and the message stores. In every branch, the goal service is modified based on the recovery solution corresponding to that branch. Once the goal service has been modified, the composition process is then simulated on the modified goal service for that branch by calling the composition algorithm recursively. For a failure, the modified goal services corresponding to all the possible recovery options are simulated in parallel. When composition completes or the user decides to abort, the goal service corresponding to one of the branches is selected.

The choice of the goal service to be presented to the user depends on how many goal services were successfully composed. If only one of the goal services was composed without failures, that service is presented to the user. If all the goal services have failures or more than one goal service has been successfully composed, the choice of the goal services is defined by user specified non-functional criteria, such as cost, reliability and historical performance.

The computation process followed can best be described in the form of a tree, with every failure occurrence denoting a node in the tree. When a failure occurs in the composition, the algorithm creates a copy of the goal service, the component and the choreographer messages stores for that node and modifies the goal service for that node to recover from the failure. Starting from that node, the algorithm creates a branch of computation for every possible recovery option from that failure, after marking that node as failed. For each of those branches, the algorithm then simulates the composition process on the corresponding modified goal service for that branch. If a branch of computation does fail at some later transition, corresponding computation branches are created for that node in a similar manner. For performance reasons,

we limit the depth of this computation tree to three failures, but it is possible to customize this depth based on performance requirements. At the end of the simulation, the algorithm selects a goal service which has not been marked as failed. If no such goal service is available, then one of the goal services is selected based on user specified criteria, as mentioned above. If different variations of the goal service are possible, one of those variations is selected based on user specified criteria.

A sample computation tree has been illustrated in the Figure 4.1. The composition process starts and fails for the first time at node n_1. Three branches are created from this node, each branch representing one solution to the failure at node n_1. Two of these solutions fail during the simulation, but node n_3 indicates that the modified goal service corresponding to the second solution is successfully composed. There are two possible solutions to the failure at node n_2 and both of them are explored, as shown. Node n_8 indicates that the goal service modified at node n_6 results in failure. This branch is marked as failed since there have been three failures in this branch.

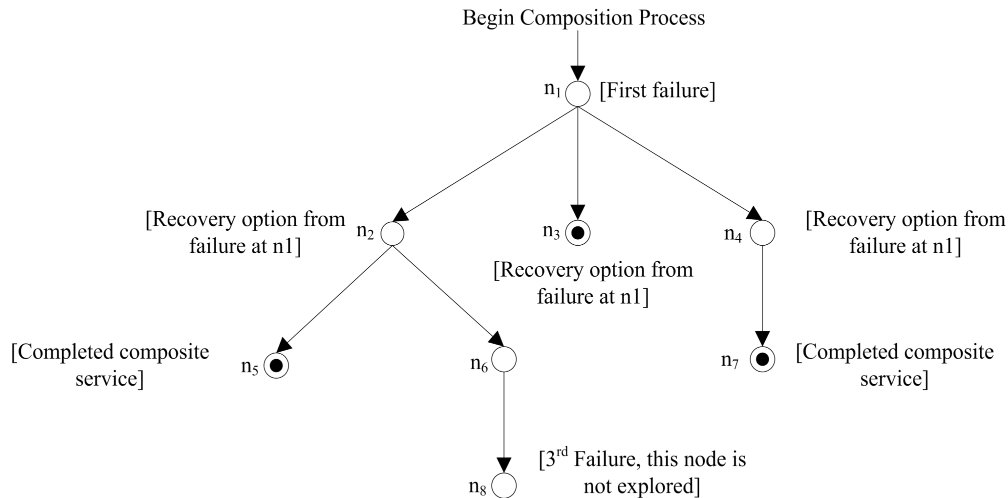

Figure 4.1 Computation tree in a failure scenario

The following scenarios would result in failures during the composition process:

1. A guard condition cannot be examined as variables required for the condition are not available

2. Message that has to be sent is not available, that is, variables required for an output action are not available

3. Variable required for input action in a component is not available or the input message set required for a function invocation cannot be provided

4. A required function cannot be completed, that is, no component provides a function specified in the goal service

5. An input action is being processed and the message set required for the input action is not available and the message header cannot be provided via flow links

When a failure is encountered, the algorithm creates a node in the computation tree and marks this node as failed. It then identifies the number of possible solutions to this failure and creates a branch for each of those solutions starting from that node. The goal service is modified for each branch based on the recovery solution corresponding to that branch, followed by a recursive call to procedure GENERATE.

Scenario 1: Variables used in guard conditions are not available

This case corresponds to the failure of a guard condition because the choreographer store does not have the variables required for the guard condition. This can happen when a variable that has not yet been encountered during composition is used in a guard condition.

There are two possible recovery options from this failure:

(i) to create an input action from the client asking for the missing variables or

(ii) to ensure that one of the component services sends a message containing the missing variables to the choreographer

We first attempt to resolve this failure using the second option, since it does not involve input from the client. In this case, the algorithm attempts to insert transitions in the goal service in a way that ensures that the variables used in the guard are available to the choreographer store when the guard condition is encountered during the composition process. These transitions are based on function invocations and output actions in the component services.

For each missing variable in the guard condition, the algorithm searches for a component T_i that has a function $funcName(I;O)$ which provides the variable as output. If multiple components house such a function, we consider each of those components in a separate branch of computation, as described above. For each such component, if the input message to that function is in the chorographer store R, a transition on that function invocation is created in the suggested goal service. If the input message to that function is not available, the same process is repeated for every variable in the input message for that function. This results in a series of transitions on function calls that provide the missing variables followed by a transition on function call $funcName(I;O)$. When the composition process is continued on this modified goal service and a transition on $funcName(I;O)$ is encountered, the algorithm will find the component that houses the function and carry out the function invocation by traversing that component service. During simulation, when the composition algorithm encounters the earlier failed guard transition, it will have the missing variable in one of the component stores. The algorithm will then search that component for an output action that has the missing variable in its message set. This corresponds to the passing of a message from that component to the choreographer, making the missing variable available to the choreographer and avoiding the failure.

The first option involves the insertion of a transition in the goal service with an input message on the missing variable. This is equivalent to getting the variable as part of an input message from the client. We undertake this option only if none of the component services can provide for the missing variable.

Scenario 2: Variable required for an output action is not available

In this scenario, the composite service has to pass an output message to the client but is unable to do so as it does not have the output message set. This happens when the algorithm encounters an output action in the goal service and the choreographer store does not have the output message.

There are two ways to recover from this failure. The first option is to search the available component services for a function call which has the missing message in its output. If some

component does have such a function, a transition on this function is inserted in the goal service. When the composition process encounters this function invocation in this modified goal service, the algorithm will traverse the component that has this function invocation and will place the missing message set in the component message store. When the failed output action is encountered, the algorithm will find the component store that has the required message set and will parse that component till it encounters an output action with the required message set. This output action is then placed in the choreographer.

The second option is to place an input action with the missing message set in the goal service. This means that the client provides the missing message set.

Scenario 3: Variable required for input action in a component is not available or the input message set required for a function invocation cannot be provided

An input action might be encountered when parsing a component service. This means that the choreographer has to pass a message to the component, which might fail if the choreographer does not have the required message set to pass on to the component. This scenario also includes the case where a function is to be called and the message set required to call the function is not available.

The resolution to these failures is similar to the resolution of failures in scenarios one and two. We explore the available components and find function invocations with output valuations resulting in the missing message set. When the goal service is modified by inserting transitions on these function invocations and the composition algorithm later simulated on this modified goal service, an attempt is made to find output actions on the missing variables from the available components. These output actions are placed in the choreographer.

Scenario 4: No component provides a function specified in the goal service

In this case, the goal service has a function call and none of the component services provide a transition on this function.

To recover from this failure, we search for a semantically equivalent function with the same input and output messages as the required function. The transition on the missing function in the goal service is then replaced by a transition on this semantically equivalent function.

Scenario 5: An input action is being processed and the message set required for the input action is not available and the message header cannot be provided via flow links

The input action referred to in this scenario is an input action in a component service. When a component is being traversed and an input action is encountered for which the message set is not available, the composition algorithm normally explores flow links to find an output action with the missing message set. If such an output action is not available, the composition fails.

To recover from this failure, we find a function invocation with an output valuation on the missing message set, in a manner similar to scenario one. The goal service is then modified by inserting into it a transition on such a function invocation. When this modified goal service is simulated, the composition process will search for an output action on the missing message set and insert it into the choreographer. Thus, when the composition process reaches the original failed transition, the choreographer store R will have the missing message set and flow links will not have to be explored.

The code snippets corresponding to the failure scenarios which have been described above are shown in Appendix B.

CHAPTER 5. CASE STUDIES

This chapter illustrates the complete web service composition, failure analysis and recovery process using two examples, namely eLibrary; the library book reservation system and the SportsReserve Service, a Stadium reservation system.

The two case studies illustrate the behavior of the composition algorithm in two capacities. The library book reservation system describes a scenario in which composition fails given the available set of components, but from which recovery is possible. This example describes the modifications made to the goal service to avoid these failures and the way in which simulation proceeds after the goal service has been modified.

The Stadium Reservation service is a scenario where the composition process fails and cannot be recovered given the available component services.

5.1 Library Book Reservation System

The library book reservation system requires three main functionalities; book searches, book delivery requests and book reservations. The goal of the system is to allow a library member to search through the library catalog for a book based on parameters such as book title, the author and the ISBN number of the book. If the library has copies of the book, the system checks if a copy is available to be checked out. If it is, the system places a request for delivery of the book to the member's home address, which is stored in the member's account information. If all copies of the book have been checked out, the system places a hold request on the book.

The web service developer is assigned the task of composing a goal service that realizes the above mentioned requirements. The goal service can be composed from five available com-

ponent services: *Availability*, *BookReservation*, *DeliveryRequest*, *MemberAddress* and *SearchBook*. The *availability* service accepts a book title and the date and checks if a copy of the book can be checked out on the entered date. *BookReservation* accepts the book title, the date and the member ID and reserves a copy of the book for that member on the specified date. *DeliveryRequest* places a request for delivery of a book to the address specified in the member's account. *MemberAddress* accesses the member's account details and returns the member's home address. The *SearchBook* service searches the library catalog for a book given the title, the name of the author or the ISBN number.

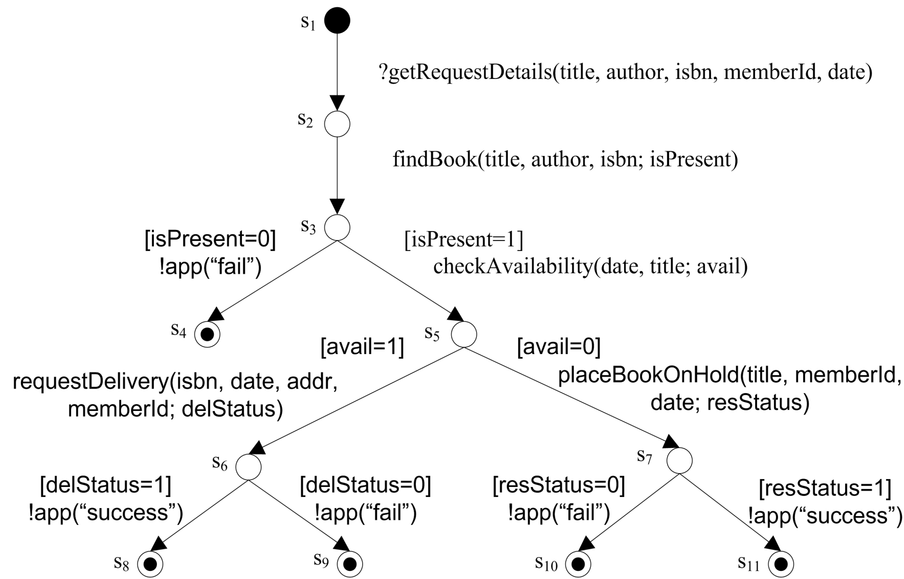

Figure 5.1 e-Library: initial goal service

The developer is assumed to be unaware of the exact nomenclature used in the component services. The developer will therefore have to give an abstract specification of the goal service. Given the requirement specification for eLibrary, the developer creates an initial goal service as shown in Figure 5.1. This goal service first accepts the variables title, author, ISBN number, date and the library member ID from the client using the input action *?getRequestDetail(title, author, isbn, memberId, date)*. The next step for the developer is to search the library catalog for the book. The developer does this by invoking a function *findBook(title, author, isbn; isPresent)* in the next transition in the goal service. The findBook function takes in three

input parameters *title*, *author* and *isbn* and has one output value *isPresent*.

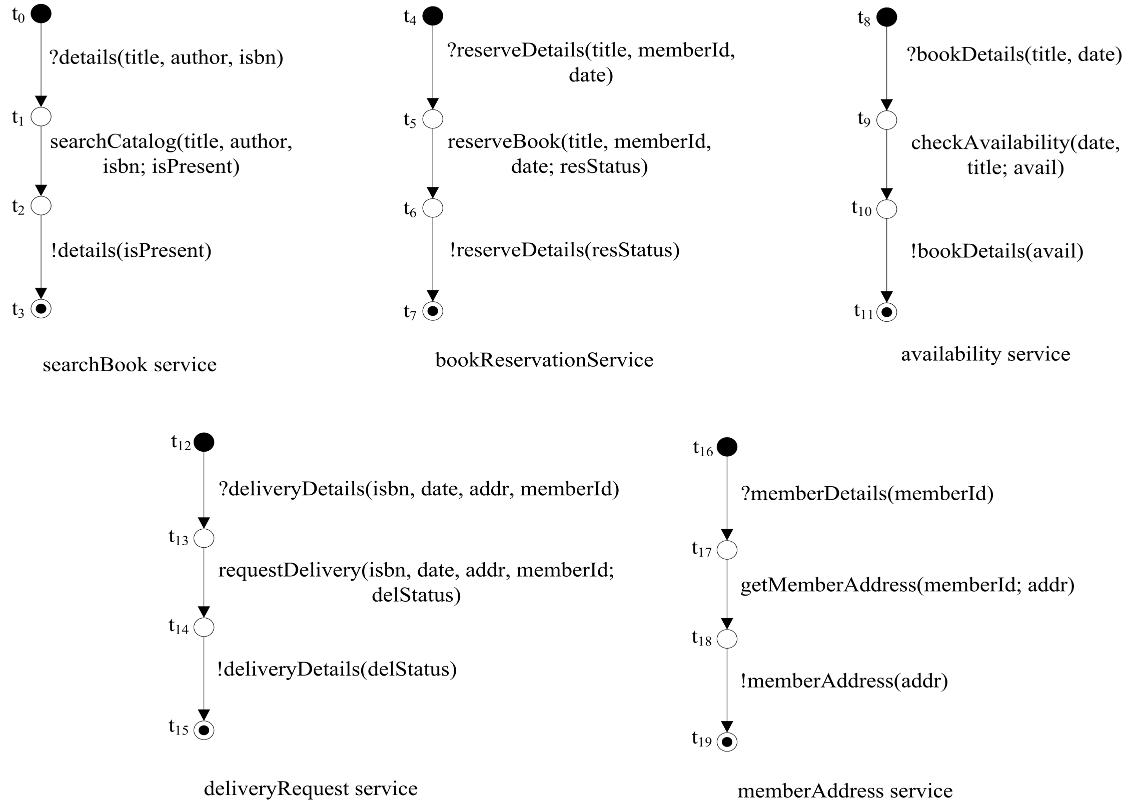

Figure 5.2 e-Library: Component Services

If the book is not present in the library as indicated by the value of *isPresent*, the application halts. If the book is present, the developer checks if a copy of the book can be checked out using the *checkAvailability* function. This function accepts a date and the title of the book and returns an output value of avail. The value of avail is 1 if a copy of the book can be checked out, otherwise avail is set to 0. If a copy can be checked out, the developer places a request for delivery of the book using the requestDelivery function. The application then halts with a success or failure message based on the response from the *requestDelivery* function. If no copies of the book can be checked out, the developer uses the *placeBookOnHold* function to place a hold request on the book. The application then halts with a success or failure message based on the response from the *placeBookOnHold* function.

The composition algorithm accepts the goal service as specified by the developer and the

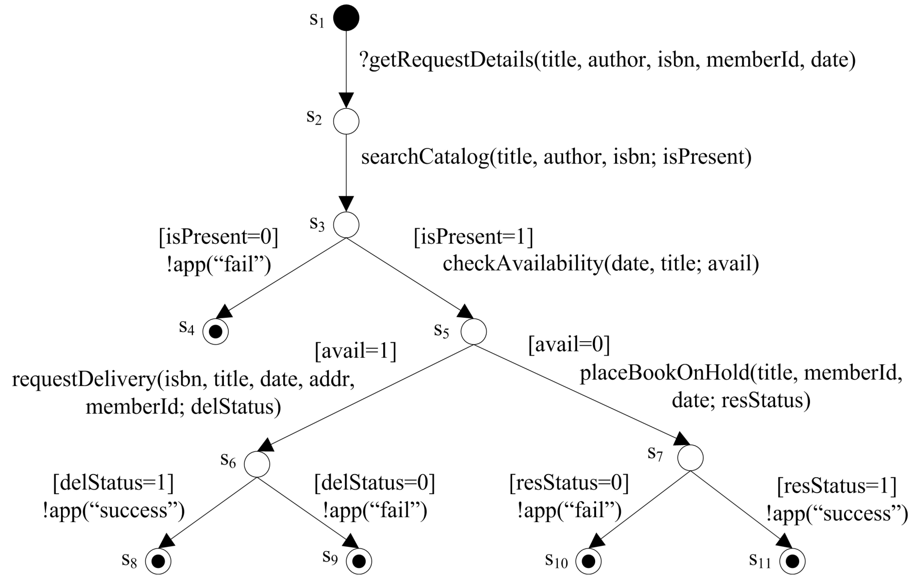

Figure 5.3 e-Library: suggested modification to the goal service after first failure

set of component services and attempts to create a choreographer for the eLibrary system. The input action *?getRequestDetail(title, author, isbn, memberId, date)* in the goal service corresponds to the receipt of a message from the client, meaning that the user has entered data in the system. The choreographer copies this input action and stores the parameters in the choreographer message store R. The next step in the composition is to create a transition that realizes the function invocation *findBook*. At this point, the process reaches case 3(b) described in section 3.2, where none of the component services are at a transition on the required function invocation. The algorithm searches for a component which has a transition on the function *findBook*, but fails at this point since none of the components house such a function. Failure recovery action is initiated at this point. The *searchBook* service has a function *searchCatalog* that is semantically equivalent to the *findBook* function in the goal service. Based on this recovery option, a separate branch of computation is created with its own copy of the goal service, the choreographer, the component services and the message stores. Since there is only one failure recovery option, only one branch is created. All the computation is now carried out in this branch. The goal service is modified by replacing the *findBook* function call with a function call to the *searchCatalog* function. The composition process is then simulated on this

modified goal service. A transition on the input action *?details(title, author, isbn)* is created in the suggested choreographer. The goal service as modified after the first failure is shown in Figure 5.3.

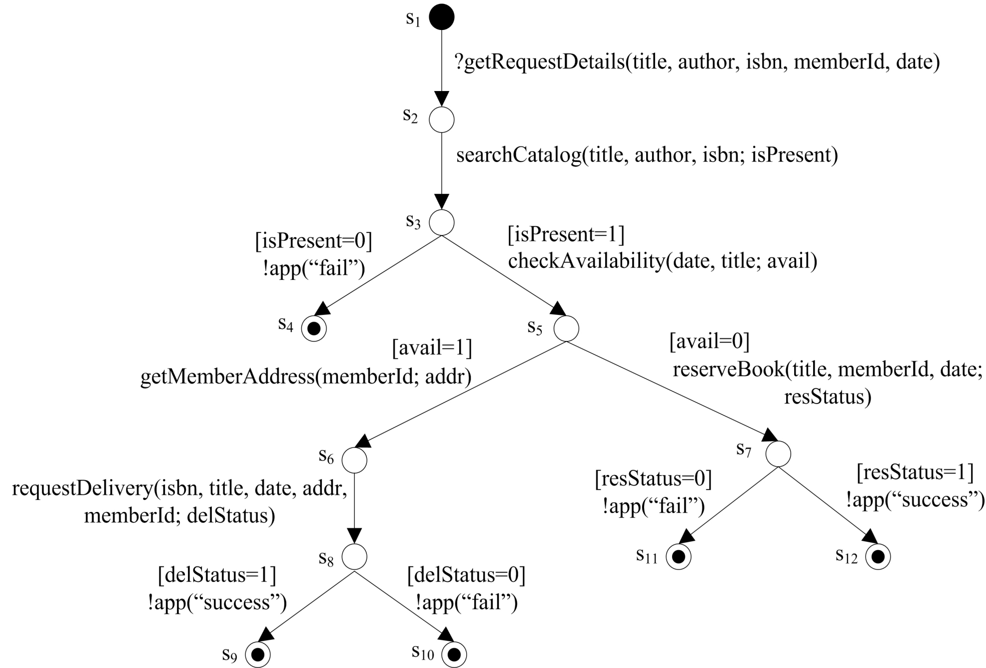

Figure 5.4 e-Library: final suggested goal service

Composition proceeds without any failures till the algorithm reaches state s_5. The composition algorithm processes each transition in a branch sequentially. It first sees the function call on *requestDelivery(isbn, date, addr, memberId; delStatus)* and fails. This is because it cannot provide the *addr* variable as it has not yet been encountered in the composition process. This corresponds to case 3(a) described in section 3.2. To recover from this failure, the algorithm searches the component services for a function with an output valuation on the variable addr. There is only one possible recovery option from this failure and it is provided by the component service *memberAddress*, which has a transition on a function call *getMemberAddress(memberId; addr)*. Furthermore, this component service also has an output action *!memberAddress(addr)* that provides the message to the choreographer. As before, a separate branch of computation is created and it is given a copy of the goal service, the choreographer, the component services as well as the message stores. The input variables required to invoke the *getMemberAddress*

function are available to the choreographer store R, so the algorithm inserts a transition on the getMemberAddress in the goal service for the new branch of computation.

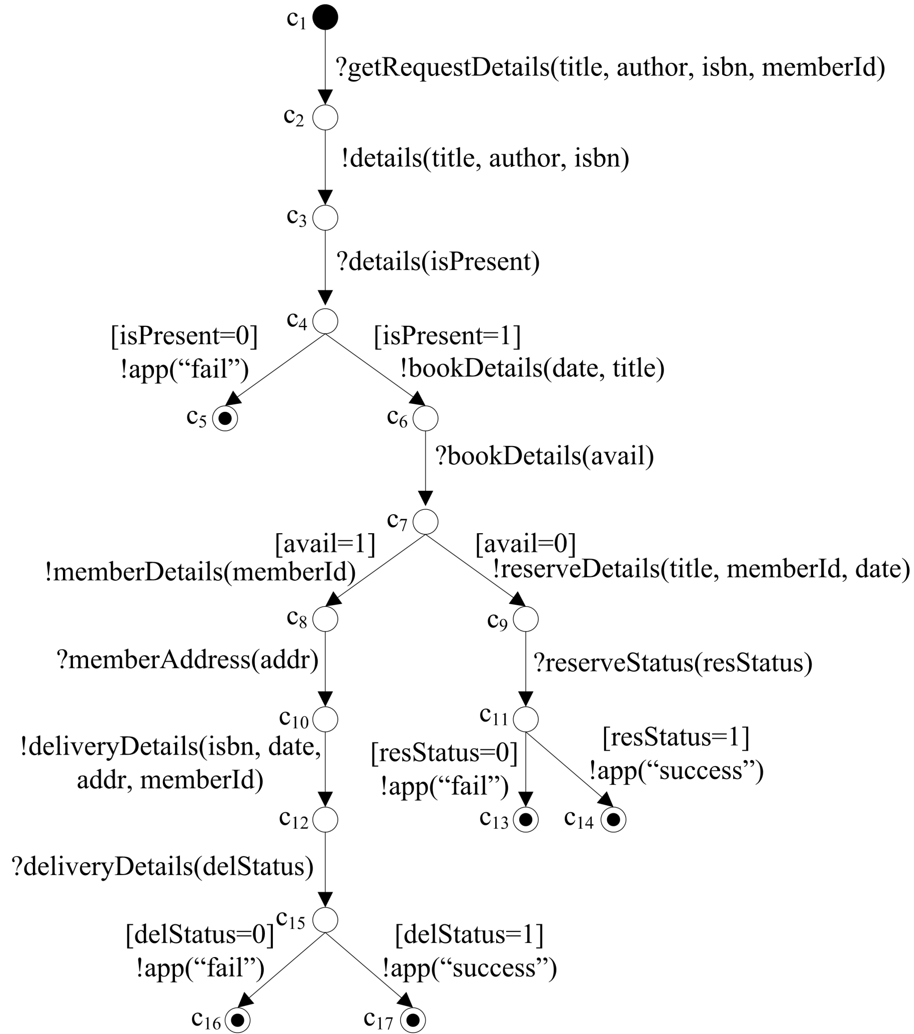

Figure 5.5 e-Library: choreographer corresponding to the final suggested goal service

To support this function invocation in the modified goal service, two new transitions on the output and input actions *!memberDetails(memberId)* and *?memberAddress(addr)* are created in the suggested choreographer. Composition of the modified goal service is then simulated in this newly created branch of computation. Since the entire message set for the *requestDelivery* function is now available, this function can now be invoked. The algorithm supports this invocation by creating actions *!deliveryDetails(isbn, date, addr, memberId)* and *?deliv-*

eryDetails(delStatus) in the suggested choreographer. This means that an output message *!deliveryDetails(isbn, date, addr, memberId)* was sent to the *memberAddress* service, the function *getMemberAddress* was invoked and the output of the function call was then sent back to the choreographer. The composition for that branch of the goal service then completes without any further failures. The composition process returns to state s_5 of the goal service to handle the transition on *placeBookOnHold*. The composition process encounters another failure here, as none of the component services have such a function. However, the *bookReservation* service has a function *reserveBook* that is semantically equivalent to the *placeBookOnHold* function. To recover from this failure, the algorithm creates a new computation branch and modifies the goal service in that branch by replacing the transition on *placeBookOnHold* with a transition on *reserveBook*. During simulation, the output action *!reserveDetails(title, memberId, date)* and input action *?reserveStatus(resStatus)* are placed in the suggested choreographer as a result of replacing *placeBookOnHold* function with reserveBook. The final suggested goal service and the suggested choreographer are shown in Figures 5.4 and 5.5 respectively.

5.2 Stadium Reservation System

The Stadium reservation system describes a typical scenario of booking a stadium. The requirements for this system are described as follows. First the user would search for a stadium of the required capacity and near a specified location. Having found such a stadium, the user would proceed to check the availability of this stadium for a given dates. If available, the user would proceed to book this stadium. If not, the system would return an error message stating that the stadium cannot be booked for the entered dates. Figure 5.6 describes the goal service for the Stadium reservation system. This case study illustrates the scenario where the choreographer cannot be composed due to lack of sufficient information.

The set of available component services include the *StadiumAvailabilityService*, *StadiumSearchService*, *CreditRecordStorageService* and the *BookStadiumService*, as shown in Figure 5.7. *StadiumSearchService* takes in the required capacity and a location and returns a stadium having capacity greater than or equal to the specified capacity within 20 miles of

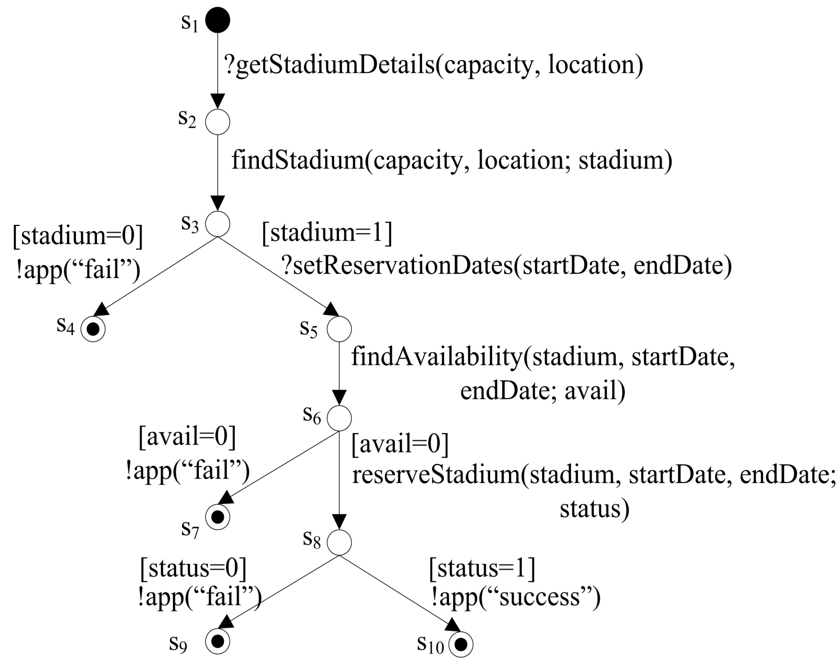

Figure 5.6 Stadium Reservation: Initial goal service

the specified location. *StadiumAvailibilityService* checks if a stadium can be reserved. *CreditRecordStorageService* accepts the credit card number and the name of the customer and stores the record for that credit card number in the system for future reference. It returns a unique *customerID* for that customer if successful. *BookStadiumService* accepts a *customerID* and reserves the selected stadium for the given range of dates. The algorithm searches for a semantically equivalent function with the same input and output parameters in the component services and finds the function *searchStadiums(capacity, location; stadium)* in *StadiumSearchService.* There is only one recovery option for this failure; the current node is marked as failed and the goal service is modified by replacing the call to *findStadium* with a call to *searchStadiums.* Composition proceeds normally on the modified goal service until the transition on *findAvailability(stadium, startDate, endDate; avail)* is encountered. Since no component service houses such a function, another failure is reported. Once again, the algorithm searches the component services for a function equivalent to *findAvailability*, and finds *StadiumAvailabilityService* which has a transition on function *checkAvailability(stadium, startDate, endDate; avail).* The current node is marked as failed and the goal service for this node is once again modified by

Figure 5.7 Stadium Reservation: Component services

replacing *findAvailability* with *checkAvailability.* Composition proceeds on this modified goal service, but encounters another failure at the transition on *reserveStadium(stadium, startDate, endDate, sponsor; status).* To recover from this the algorithm searches the component services for a function equivalent to *reserveStadium.* However, the component services which provide for stadium reservation require a sponsorID that had not been accounted for in the goal service. Hence the algorithm is unable to recover from this failure and returns the modified goal service after highlighting the failed transition. The suggested goal service and the partial choreographer for this case study are shown in Figures 5.8 and 5.9.

In Figure 5.9, the transition from state t_8 to state t_{10} has been grayed out to indicate that it will not be included in the partial choreographer.

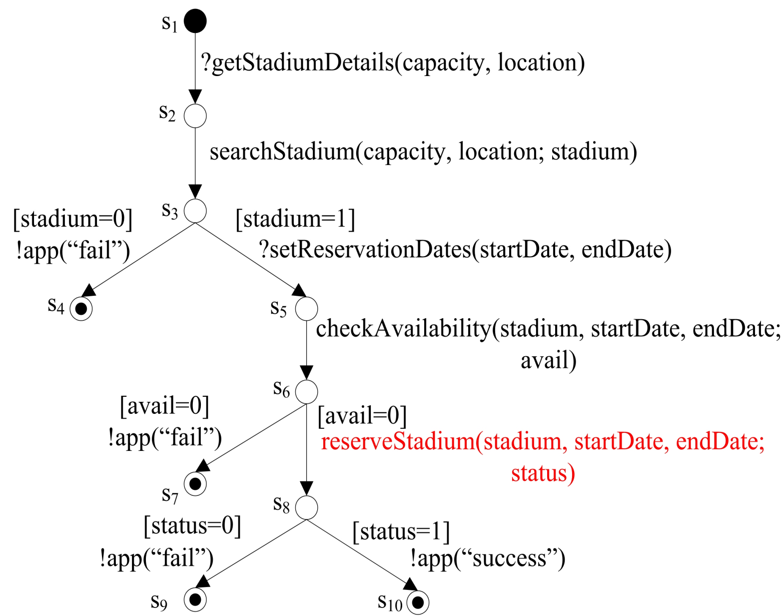

Figure 5.8 Stadium Reservation: Suggested goal service with failed transition

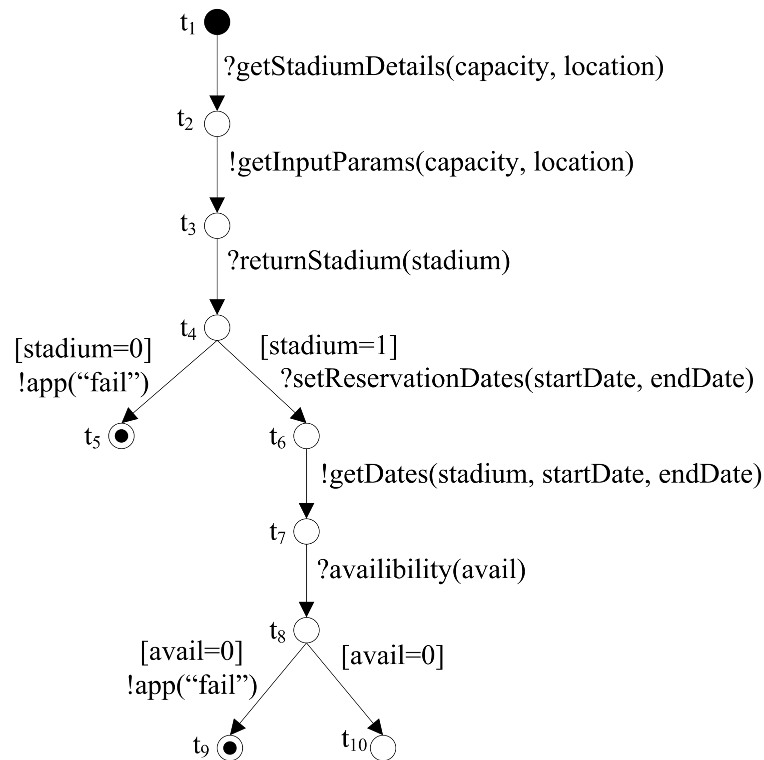

Figure 5.9 Stadium Reservation: Partially generated choreographer

CHAPTER 6. CORRECTNESS AND COMPLEXITY ANALYSIS

This chapter describes the correctness and the complexity analysis of the composition algorithm and the failure analysis and recovery module.

We first discuss the correctness of the composition algorithm by itself. Given the initial goal service and the component service, we show that the algorithm always terminates, either in failure or with a complete choreographer. We then proceed to describe the correctness of the failure analysis module. When failure occurs, suggestions are made to the goal service to recover from failure. We show that the process of finding recovery solutions to a failure always terminates for all the failure scenarios; and if possible, returns a suggestion to the goal service that avoids the failure. This is followed by a section on the complexity analysis of the composition algorithm.

To facilitate the recovery from failure, we have made some assumptions about the component services. These assumptions are listed below:

(i) Every component service has at least one input action and one output action. Since every component is an independent entity, the component must have at least one input action to accept input messages. This serves to provide data to the component service. The component service will send the result of its computation through an output action, and hence must have at least one output action. Under this assumption, a component service minimally has two transitions, a transition on an input action and a transition on an output action.

(ii) The input to a function is provided by means of an output action by the calling entity or is obtained by the results of earlier function invocations in the same component service. In the case where the choreographer calls a function in a component service, the choreog-

rapher must send a message containing the input parameters to the component service. Otherwise, the function call in the component service must be preceeded by a number of function invocations, which provide the input parameters as their output values.

(iii) If a component service provides an output value that was obtained as a result of some computation within the component service, the output action that provides the output message must be preceeded by one or more function calls that provide the output message as their result.

6.1 Correctness

Claim: Given a goal service T_g with start state s_0 and n component services $T_1,\ldots,T_n$ the procedure *GENERATE* always terminates with a successful composite service T_{cr} that is equivalent to the goal service or provides a partial composite service in the event of composition failure.

Proof Sketch: The composition algorithm processes the goal service one state at a time, managing the transitions in the choreographer based on the transition in the goal service. The recursive algorithm terminates when all the states in the set of final states of the goal service s^F are processed. Thus the composition process always terminates.

The proof of equivalence is stated using contradiction. There are four possible cases:

(i) for an input action in T_g there is no corresponding input action in T_{cr}

(ii) there is no output action in T_{cr} corresponding to an output action in T_g

(iii) a function invocation in T_g is not modeled by T_{cr}

(iv) certain transition sequences in T_g cannot be provided by T_{cr} because some guards are not available

Case (i) This situation cannot be true, because an input action in the goal T_g is directly copied into the choreographer.

Case (ii) For every output message in T_g, a corresponding output action is placed in T_{cr} if the required message is present in the choreographer store R. If the message is not available in R, the component stores are searched for this message. If the *msgSet* required for the output message is available in the component stores, the algorithm proceeds to find output actions in the component services that can get the *msgSet* into the choreographer store. The component stores are finite and every component service has a finite number of transitions. After parsing all the required output actions from the component services, the choreographer store R will get the entire *msgSet*. At this point, the output action is created in the choreographer. If the *msgSet* is not provided by the component stores as well, the composition step halts in failure. Hence, case (ii) cannot be true.

Case (iii) This case relates to function invocations. Function invocations in the goal service are handled by creating a series of transitions in T_{cr} which interact with the component services so as to provide for the function invocation. Such a sequence must include transitions that send messages containing the input parameters to the component service that provides the function as well as transition that accepts the message from the component service if the function has an output value. For this function to be invoked, the choreographer store R must contain all the input parameters required for the function. If the input *msgSet* to this function is not in R, the component stores are searched for the input parameters. If the component stores contain the *msgSet*, the corresponding component services are traversed to find output actions that can put the function input parameters into R. Once the component services are parsed, the sequence of transitions required to realize the function invocation are placed in the choreographer. If the input parameters are not available in the component stores, composition stops with a failure. Also, if no components provide the function, the composition halts with a failure. Thus the choreographer models a function invocation in the goal service.

Case (iv) Whenever a guard condition is encountered, the algorithm checks if it can provide all the required variables for the guard condition. If the required variables are present in R, the guard condition is placed in the choreographer. If R does not contain all the variables, the component stores are searched for these variables. If the component stores provide the

variables, the corresponding component services are traversed for output actions that put the variables into R. If the component stores do not provide the variables, failure is reported. Once the choreographer has all the variables required to realize the guard condition, the choreographer creates the transition corresponding to the transition in the goal service. Thus, guard conditions are always realized if the choreographer can provide the variables required for the guard condition.

Claim: Given a goal service T_g with start state s_0 and n component services $T_1,\ldots,T_n$, the failure analysis module always terminates. Also, for a given failure scenario, if it is possible to recover from the failure, the changes that are suggested to the goal service as part of recovery ensure that the failure is avoided in subsequent composition on the suggested goal service. If recovery is not possible, the algorithm terminates in failure.

Proof sketch: We consider the proof in terms of the five failure scenarios as described in Chapter 4. There are five scenarios:

(i) Variables used in a guard condition are not available

(ii) Variables required for an output action are not available

(iii) Variables required for an input action or the input message set of a function invocation are not available

(iv) A function invocation cannot be provided

(v) An input action in a component service is being processed and the message set required for the input action is not available and the message header for the input action is not provided by the flow links

We consider the first case. If variables for the guard condition are not available, the algorithm attempts to search for output actions in the component services which can provide the required variables to the choreographer. For each variable in the guard, a component service must send a message to the choreographer containing this variable so that the variable becomes available in R. To ensure that this component service sends such a message to the

choreographer, a function invocation is searched in the component service that provides an output value of the missing variable. Failure recovery in this case suggests the creation of a transition on this function call in the goal service. When composition proceeds on the suggested goal service, the choreographer will send a message to the component service to invoke the function and will get an output message from the component service which will provide it with the missing variable. If the choreographer store cannot provide all the input parameters for this function call, the process is recursively invoked for every input parameter of this function. Thus, the failure recovery process in this case would result in the creation of a sequence of transitions in the goal service that will provide the missing variables for the guard condition. This process always halts, since the number of recursions is bounded by the number of transitions in the component services, which is finite. Also, the suggestions made to the goal service ensure that the variables for the guard condition are available in R, avoiding the failure. The proof of termination and correctness for cases (ii),(iii) and (v) is similar to that of case (i).

To recover from the failure in case (iv), the algorithm searches the component services for a semantically equivalent function that can provide for the missing function invocation. For every component service that provides an alternate function, a suggestion is made to replace the function that is unavailable in the goal service with the equivalent function provided by the component service. Since the functions are semantically equivalent and have the same input and output parameters, the replacement does not affect the functionality of the goal service. Also, the replacement function can be provided by the component services, which avoids the failure. Since there are a finite number of component services, this process always halts.

6.2 Complexity Analysis

The complexity analysis of the entire composition process is described in terms of the complexity of the composition algorithm and the complexity of the failure analysis scenarios. We first consider the complexity of the composition algorithm by itself. We then determine the complexity of the failure analysis and recovery modules for each of the failure scenarios.

Then we describe the complexity of the overall process.

The complexity of the composition algorithm is determined by the number of recursive calls made to function GENERATE while traversing the goal service from the start to the final states. Let $|S_g|$ be the number of states in the goal service. Let there be n component services, each component service having $|S_c|$ number of states. During each recursive call, the goal service is at a particular state and corresponding to this state in the goal service, the component services are at particular states. Each of the n component services can be at any one of their $|S_c|$ number of states. Since there are n component services, there can be $|S_c|^n$ number of combinations of states in the component services that can be associated with the current state of the goal service. There are $|S_g|$ number of goal states, hence there will be a total of $(|S_g| \times |S_c|^n)$ number of possible combinations of states. Each combination is associated with constraints over variables in the choreographer and the component stores. By constraints, we mean that every combination of states can be associated with a guard condition, that uses a number of variables from the choreographer and component stores. Let k be the total number of constraints over variables in the choreographer and the component stores. Variable constraints are updated when the current states of the goal and component services are changed. Hence the complexity of the composition algorithm itself is $O(|S_g| \times |S_c|^n \times 2^k)$.

We now consider the complexity analysis of the failure scenarios. For each scenario, the complexity is defined in terms of the number of steps required to identify the number of possible solutions and for each solution, the steps required to suggest changes to the goal service.

Consider scenario 1, where the guard variables required for a transition in the goal service are not available. To recover from this failure, the algorithm attempts to find component services which can provide the variables in the guard condition to the choreographer. Hence, for every variable in the guard condition which is not available in R, the algorithm searches for component services which contain a function having an output value of that variable. Let the number of variables in the guard condition be represented by $vars(g)$. For every $v \in vars(g)$, all the component services are searched for the function which provides the variable in its output. In the worst case, all the transitions in the component services will be

searched, which is $O(n|S_c|^2)$. For every selected component that provides this function, the algorithm travels to the transition on the function, in the worst case, this requires the traversal of $O(|S_c|^2)$ transitions in the component service. The choreographer store R must have the input parameters required to call this function. If it does not, this process is recursively called on the input message set of the function. Since the component services have at most $O(n|S_c|^2)$ transitions, the component services can provide a maximum of $O(n|S_c|^2)$ functions. Hence, the maximum number of recursive calls that can be made is $O(n|S_c|^2)$. For each recursive call, the same steps as listed above are repeated, hence the number of steps required in each recursive call is given by $O((n+1)(|S_c|^2))$. Hence, the total complexity required to suggest a goal service for a single v is $O(n^2|S_c|^4)$. The same steps are repeated for every $v \in vars(g)$, hence the total complexity of this scenario is still $O(n^2|S_c|^4)$.

The handling of failures in scenarios 2, 3 and 5 is similar to that of scenario 1, in that the failure is caused due to missing variables and the algorithm attempts to suggest changes to the goal service to account for those missing variables. The complexity analysis of scenarios 2, 3 and 5 is the same as that of scenario 1.

We now consider the complexity of the failure scenario 4. When a function in the goal service cannot be provided by any of the component services, the algorithm has to scan at most $n(|S_c|)(|S_c|-1)$ transitions in the component services before finding an equivalent function. This is $O(n|S_c|^2)$. For every component service that provides such an equivalent function, we create a copy of the goal service, which will require $O(|S_g|^2)$ steps. If there are l components which provide equivalent functions, the complexity to replicate the goal service for each of those components will be $O(l|S_g|^2)$. For each such goal service, we replace the transition on the missing function with a transition on its equivalent function provided by the relevant component. The corresponding transitions on the goal service and the component service are marked during the search itself and the replacement can take place in constant time. Hence, the total complexity for failure scenario 4 is $O(n|S_c|^2 + l|S_g|^2)$.

The complexity of the entire process is the sum of the complexities of the composition process and the failure scenarios. If a failure occurs and it is possible to recover from the

failure, the composition process is repeated on the suggested goal service. After the goal service is modified based on the suggestions made to recover from the failure, the complexity of the subsequent composition on the suggested goal service will be dependent on the number of states added to the suggested goal service. Furthermore, if multiple goal services are created to explore different solution paths to a failure, the composition on these goal services is done in parallel with each other. Hence, for every failure, we need to consider the worst complexity among the complexities of the goal services suggested to recover from this failure. If it is not possible to recover from a failure, the process halts and returns a partially modified goal service and its corresponding choreographer to the user. In this case, we need to consider the complexity of the composition algorithm and failure analysis upto the point of failure.

Consider for example, a scenario where the composition fails, but it is possible to recover from failure. In this case, the initial composition of a goal service is $\mathrm{O}(|S_g| \times |S_c|^n \times 2^k)$. On failure, we add the complexity of the failure scenario to the overall complexity as well as the complexity required for the subsequent composition on the suggested goal service. If the composition failed due to missing input variables and m new states were added to the suggested goal service, the overall complexity of the composition process becomes $\mathrm{O}((|S_g| \times |S_c|^n \times 2^k)$ $+ (n^2|S_c|^4) + (|S_g + m| \times |S_c|^n \times 2^k))$.

CHAPTER 7. IMPLEMENTATION

MoSCoE provides an open source tool for composing web services expressed in terms of LTSs [2]. This tool has been implemented using Java for the core processing and uses Swing API and the Cytoscape plugin to render the services. The tool is available at www.moscoe.org. Figure 7.1 shows a screenshot of the MoSCoE tool.

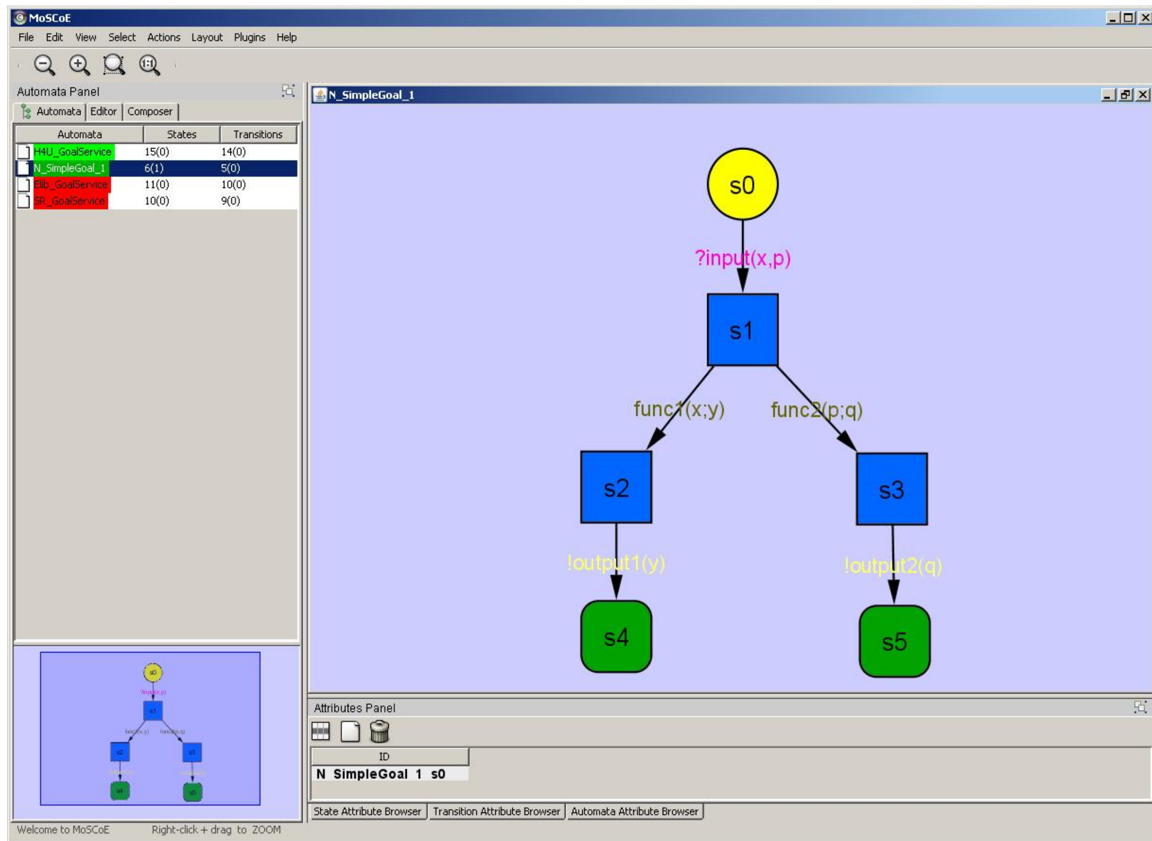

Figure 7.1 MOSCOE Service Composition tool: screenshot

7.1 Service Creation and Representation

A web service in MoSCoE is expressed in terms of an LTS. A user can create a web service using the tool itself. Future versions of the tool will provide the ability to import a web service specified in BPEL and automatically convert it to an LTS. The tool provides a set of items that can be rendered on the screen to create the goal service. It provides for the insertion of transitions as well as start (colored red), intermediate (colored blue) and final states (colored green) of the goal service. Actions and guard conditions can be inserted through the actions menu. Figure 7.2 shows the assignment of a a guard condition [x==0] on the transition from state s_1 to s_2.

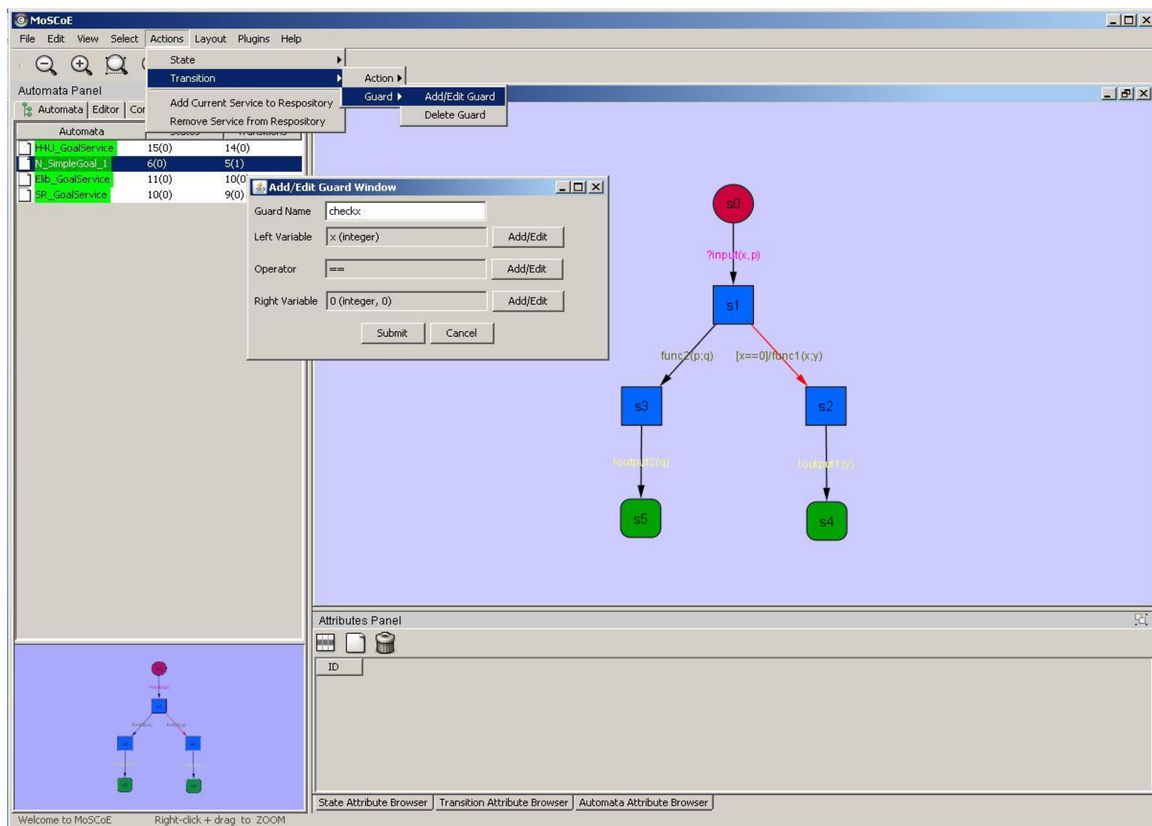

Figure 7.2 MOSCOE: Creating a guard on a transition

The composition algorithm and the failure analysis and recovery approach described in this work have been implemented in version 2 of MoSCoE. We now provide a description of the architecture of the tool.

7.2 Architecture

7.2.1 Web Service Representation

A Labeled Transition System is represented by an *Automata* object. Transitions within this LTS are represented by a list of *Transition* objects. Every *Transition* object describes the action and guard condition associated with it, via the *Action* and *Guard* objects. The states in the LTS are represented by *State*, and every *Transition* object contains a begin and end state representing the start and end states of that transition. A guard uses the *Variable* and the *Operator* objects to represent the guard condition. A UML diagram of the classes used to represent an LTS is shown in Figure 7.3.

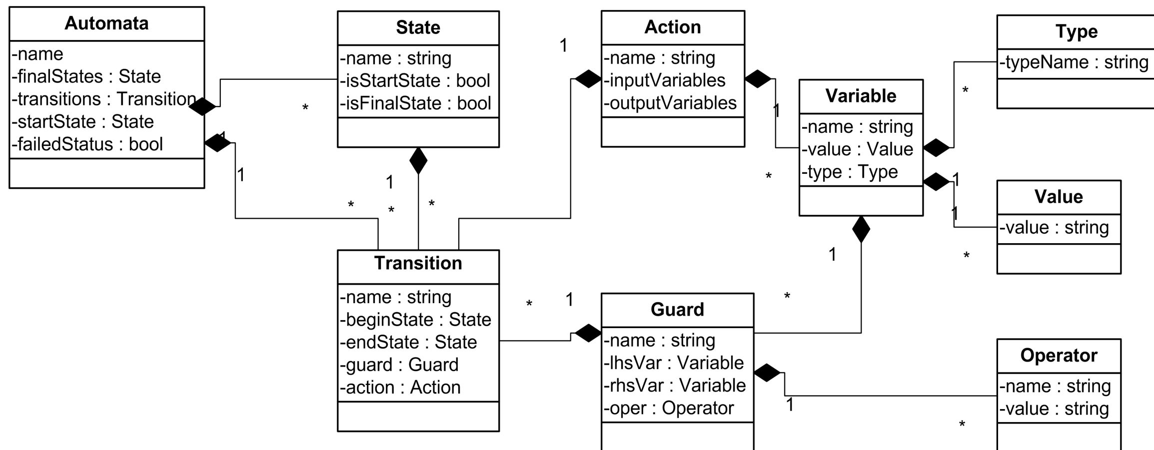

Figure 7.3 UML diagram depicting the web service data

7.2.2 Processing Module

We introduce a number of new classes to support the framework for the composition process as well as the failure analysis and recovery. These classes include the *CompositionManager*, *Composer*, *CompositionHandler*, *ServiceRepository* and *AutomataHelper*. The *CompositionManager* class manages all aspects of the composition process. It maintains a record of the number of compositions performed during an iteration by using a list of *Composer* classes. Every *Composer* object holds a goal service and a choreographer corresponding to that goal service. The Composer objects represent the nodes of the computation tree as described in chapter

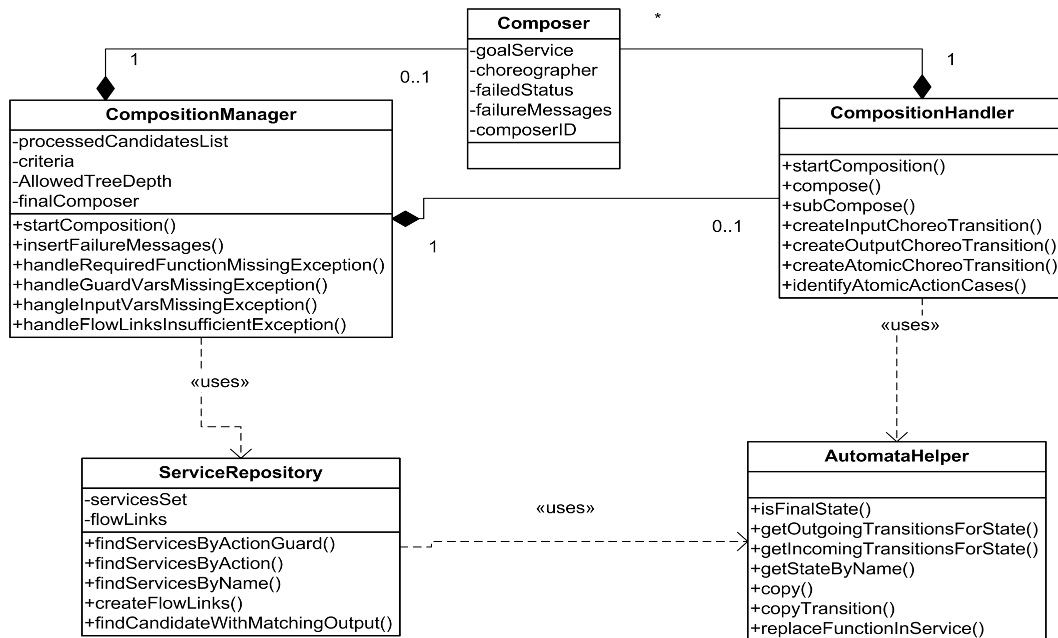

Figure 7.4 UML diagram depicting the composition manager classes

4. At the end of the entire composition process, the *CompositionManager* selects one *Composer* from amongst the list of *Composer* objects and presents it to the user. This corresponds to the selection of the best possible goal service and its choreographer. The *CompositionManager* is responsible for handling the failure analysis and the recovery options. *CompositionHandler* class contains the composition algorithm and is responsible only to compose. If a failure occurs, the CompositionHandler class propogates this failure to the CompositionManager class. When the user clicks on compose, the *CompositionManager* creates an initial *Composer* object and calls on *CompositionHandler* to begin the composition process. If a failure occurs, the *CompositionHandler* passes control to *CompositionManager* along with information about the failure. *CompositionManager* then creates the appropriate number of *Composer* objects with their own goal services, based on the failure scenario. It then calls upon the *CompositionHandler* to simulate the composition on every *Composer* object. The *ServiceRepository* maintains the list of component services and provides functions associated with the component services. *AutomataHelper* provides utility functions for traversing automata. The UML diagram for these classes is shown in Figure 7.4.

7.3 Demonstration

We now demonstrate the working of MoSCoE composition tool on the E-Library and the Stadium Reservation System which have been described in chapter 5.

7.3.1 Demonstration: E-Library case study

The user can create the goal service by drawing it in the editor panel, using the icons provided. Once the service has been created, it will be displayed in the list of available automata. This is shown in Figure 7.5

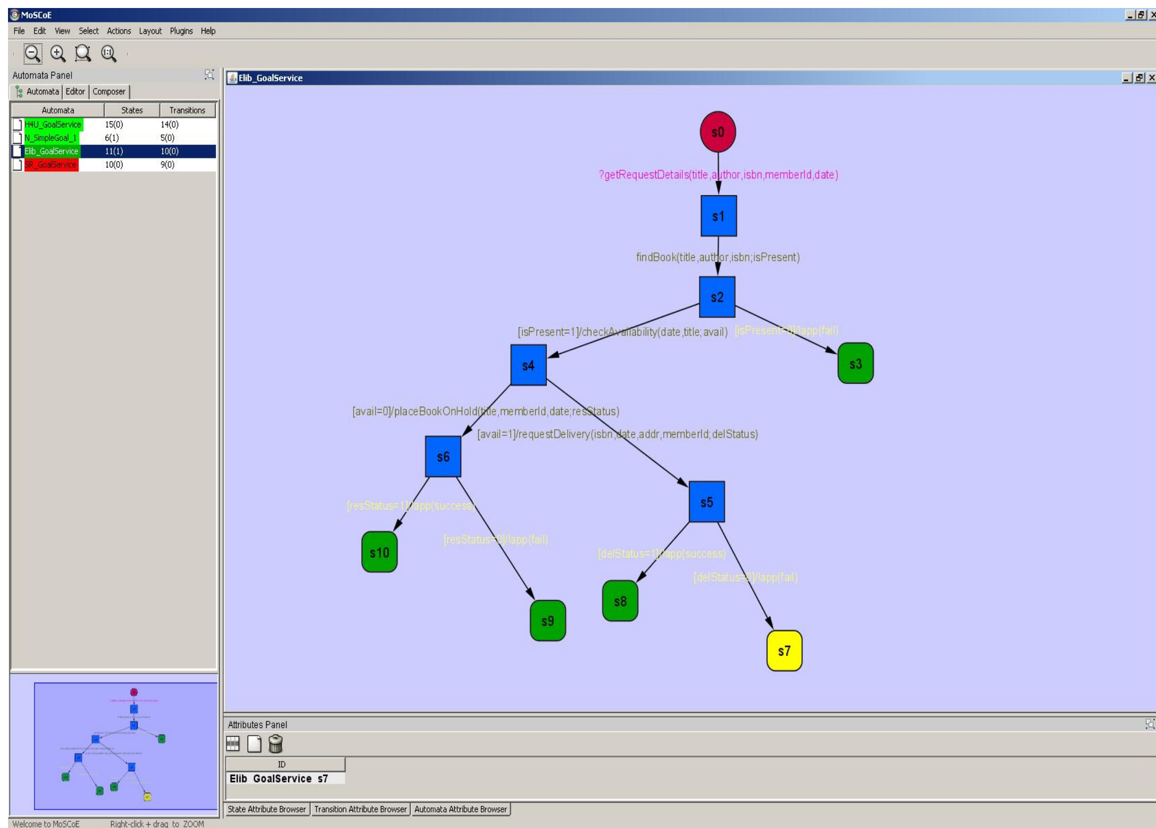

Figure 7.5 The e-Library goal service in MoSCoE

Once the goal service has been specified, the list of component services must be provided. The user can specify the services repository by selecting component services from the list of available component services. To begin composition, the user then selects a goal service from the list of goal services and clicks on compose. This is shown in Figure 7.6

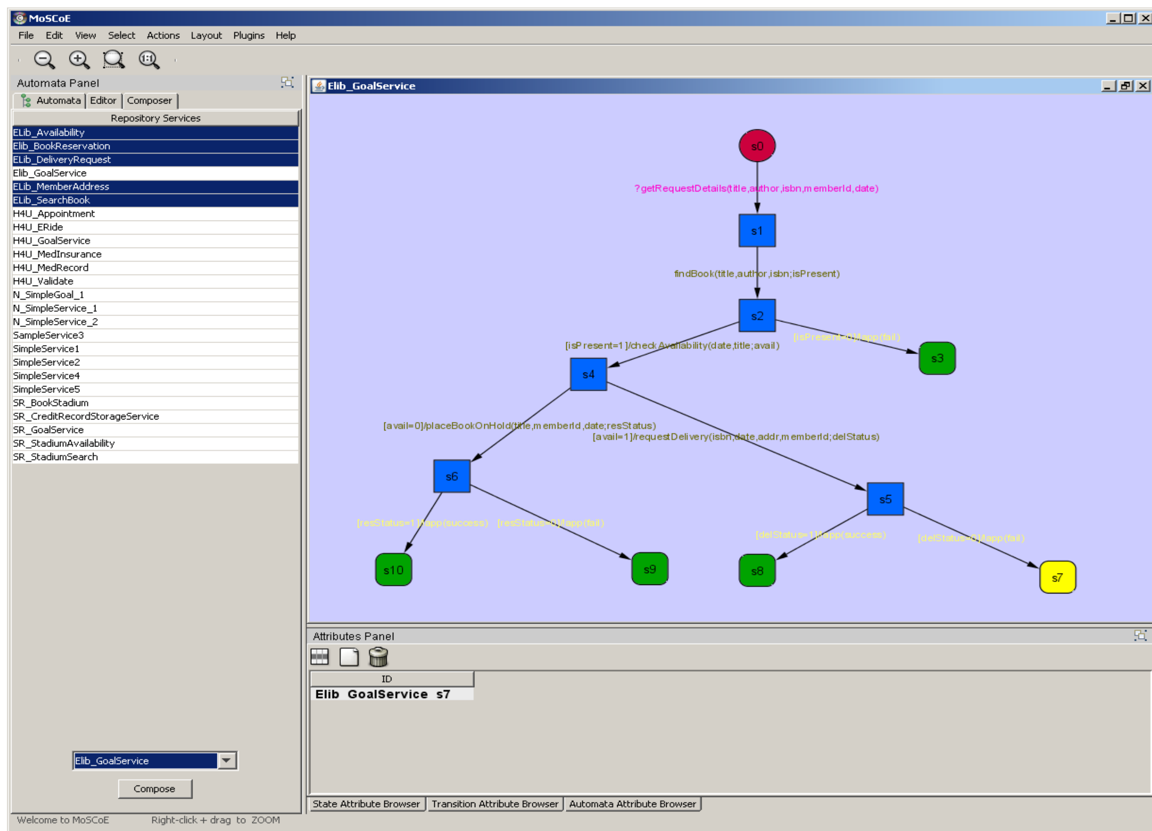

Figure 7.6 e-Library component services and goal service selected

MoSCoE begins the composition process when the user clicks on the compose button. An initial *Composer* object represents the original goal service specified by the user. This composer object also maintains a choreographer *Automata* object. The service repository provides the list of component services, corresponding to the component services selected by the user. The *CompositionHandler* always works on one Composer object at a time. Any changes to the choreographer are made to the choreographer contained in this Composer object. The *CompositionHandler* also maintains a track of the current state of the goal service, the choreographer and the component service, while composing the current *Composer* object. Whenever failure occurs, a new *Composer* object is created and a copy of the the current failed composer's goal service is provided to this new Composer. The new Composer then has its goal service modified based on the solution to the failure. The composition is then simulated on this new Composer object.

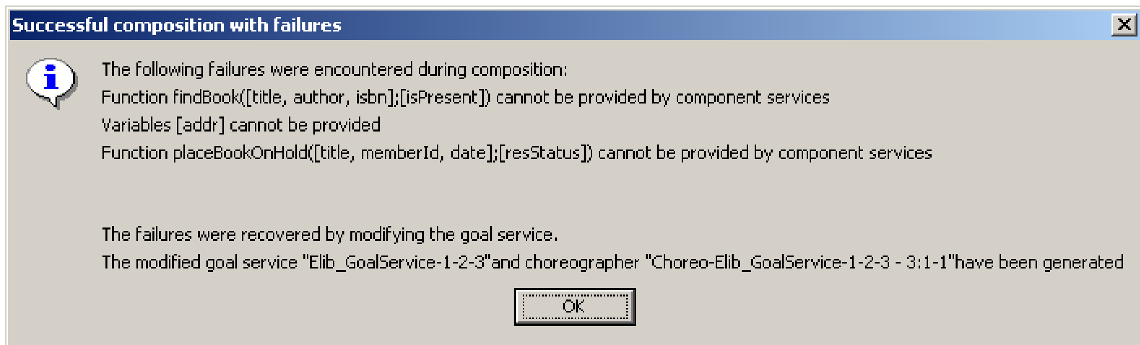

Figure 7.7 e-Library status message

Every *Composer* object has a set of failure messages which are encountered during composition. Every time a failure occurs and a new *Composer* object is created, the set of failure messages from the current *Composer* object are copied into the new *Composer* object. Failure messages related to new failures encountered while composing the current *Composer* object is then added to the list of failure messages held by that *Composer* object. In this manner,

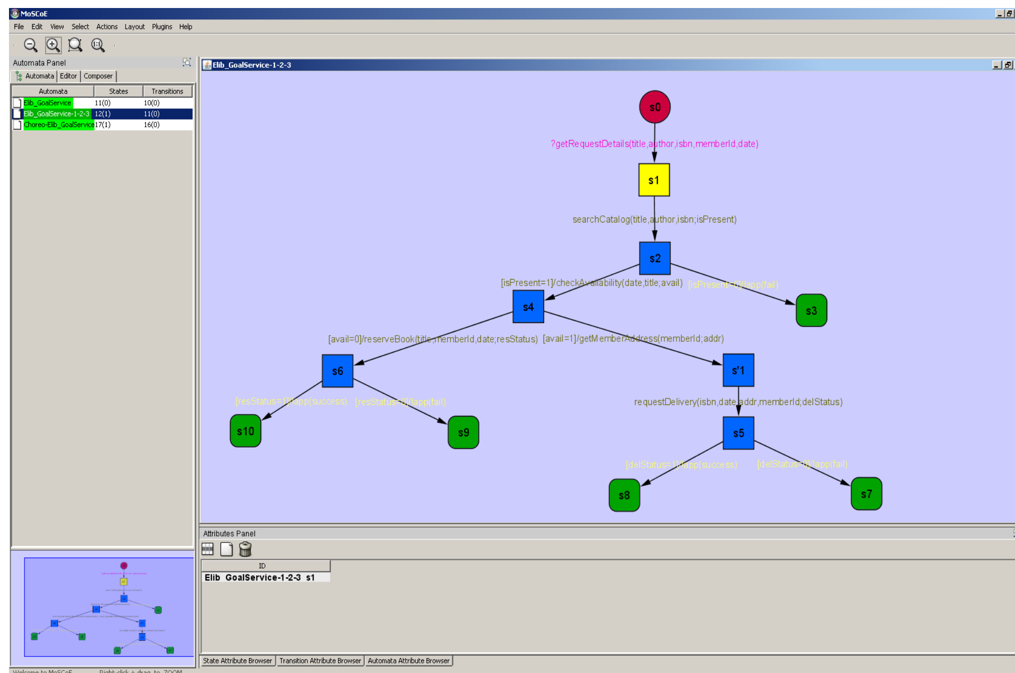

Figure 7.8 e-Library final suggested goal service

the tool maintains a record of all the failures encountered during composition. At the end

of the composition process, the tool selects the best possible composer object and provides the user with the goal service and the choreographer corresponding to this *Composer* object. The user is also given the list of failure messages held by this *Composer* object. Finally, the user is presented a status message indicating the status of the composition process. The status message for the composition of the E-Library goal service has been shown in Figure 7.7. As indicated by the status message for the e-Library system, three failures were encountered

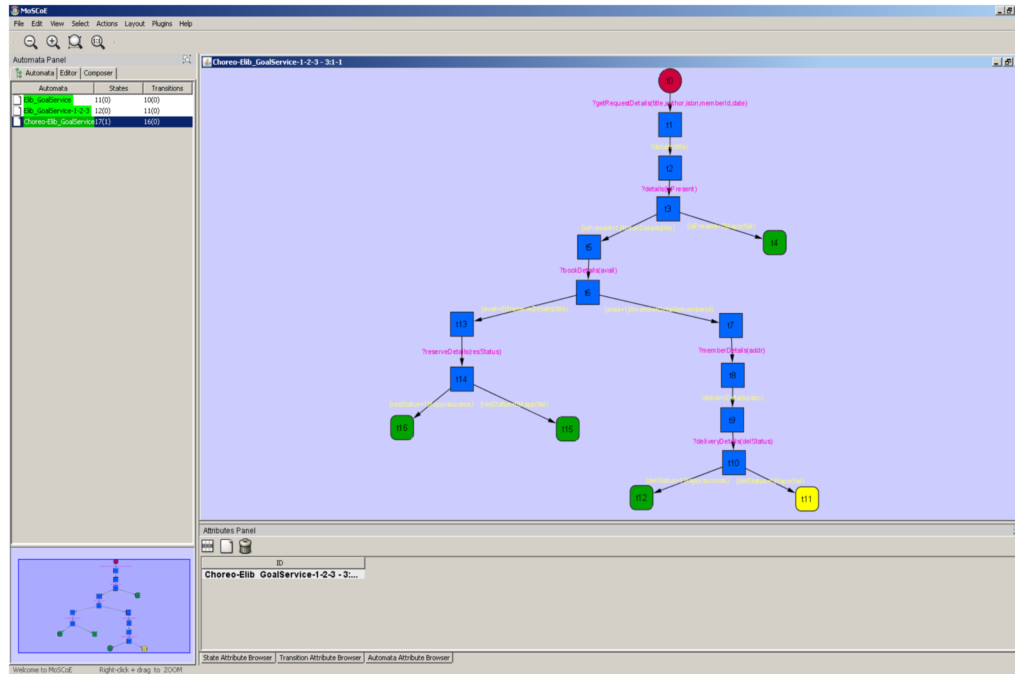

Figure 7.9 e-Library choreographer corresponding to the final suggested goal service

during composition, but it was possible to suggest changes in the goal service so as to recover from these failures. At the end of the composition process, a number of *Composer* objects with their own goal services and choreographers are available. The user is presented with the choreographer and the goal service of one of these *Composer* objects. If there are more than one *Composer* objects with goal service and choreographers that successfully composed, the non-functional requirements as specified by the user are applied to select one of those *Composer* objects. The specification of non-functional requirements is a future expansion of the MoSCoE tool and has not been implemented in this version. Currently, if there are more than

one successfully composed *Composer* objects, the first one in the list is selected and its goal service and choreographer are presented to the user. If all of the *Composer* objects failed during their respective compositions, one of them is selected and the partially modified goal service and partial choreographer corresponding to that goal service are presented to the user. The goal service which has been modified based on these suggestions and the choreographer corresponding to this goal service are shown in Figures 7.8 and 7.9.

7.3.2 Demonstration: Stadium Reservation System Case Study

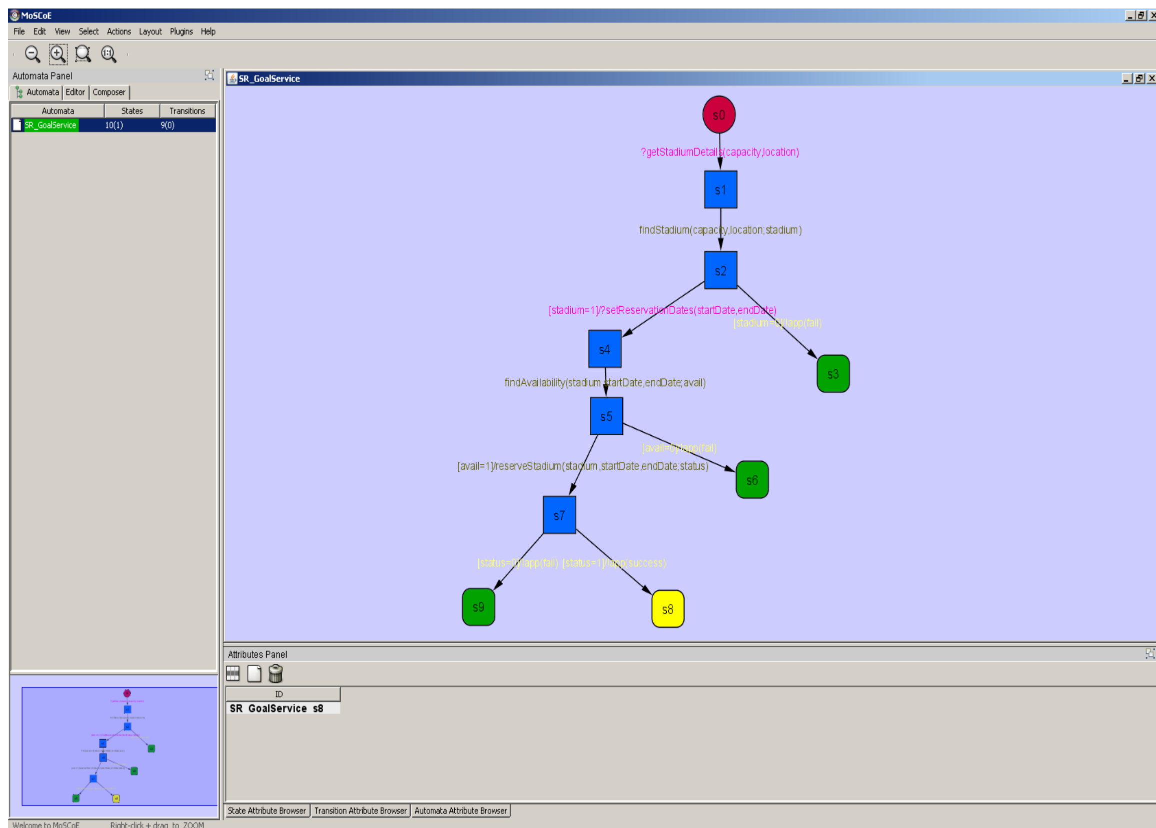

Figure 7.10 MoSCoE: Stadium Reservation goal service

The Stadium Reservation System is used to demonstrate the working of the composition process when it is not possible to recover from a failure that occurs during composition. The user specifies the goal service as shown in Figure 7.10. This is the same as the goal service illustrated in Figure 5.6.

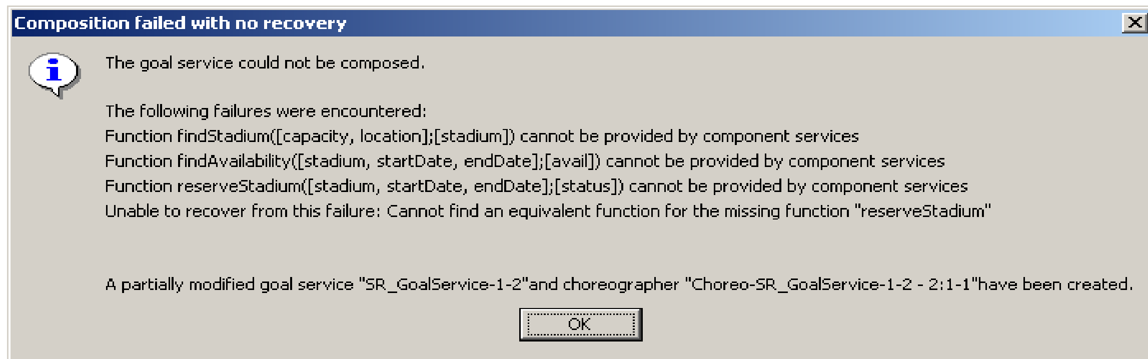

Figure 7.11 MoSCoE: Stadium Reservation composition status message

The composition algorithm fails twice initially but it is able to suggest modifications to the goal service that allow recovery from these failures. However, at the third failure, it is unable to find a function to replace the *reserveStadium* function and reports that the composition has failed and recovery is not possible. This message is shown in Figure 7.11. The suggested goal service is shown in Figure 7.12

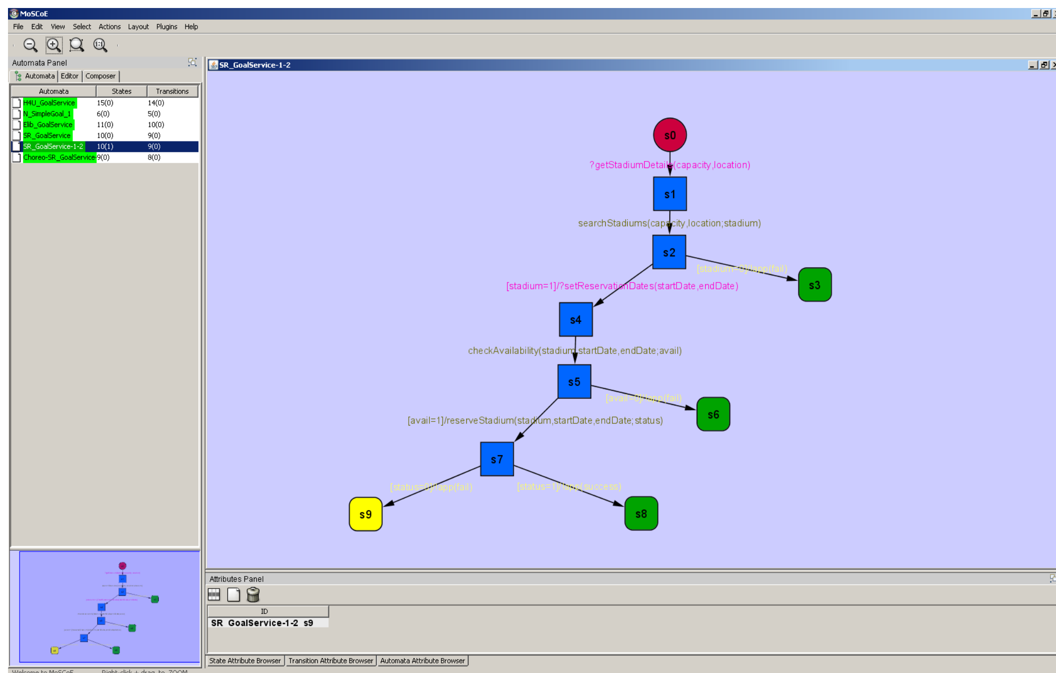

Figure 7.12 Stadium Reservation System: final suggested goal service

As indicated in the message, the MoSCoE tool was able to recover from the first two failures

by suggesting changes in the goal service. It was unable to recover from the third failure and returns the partially modified goal service and the partial choreographer corresponding to this goal service. This has been shown in Figure 7.13

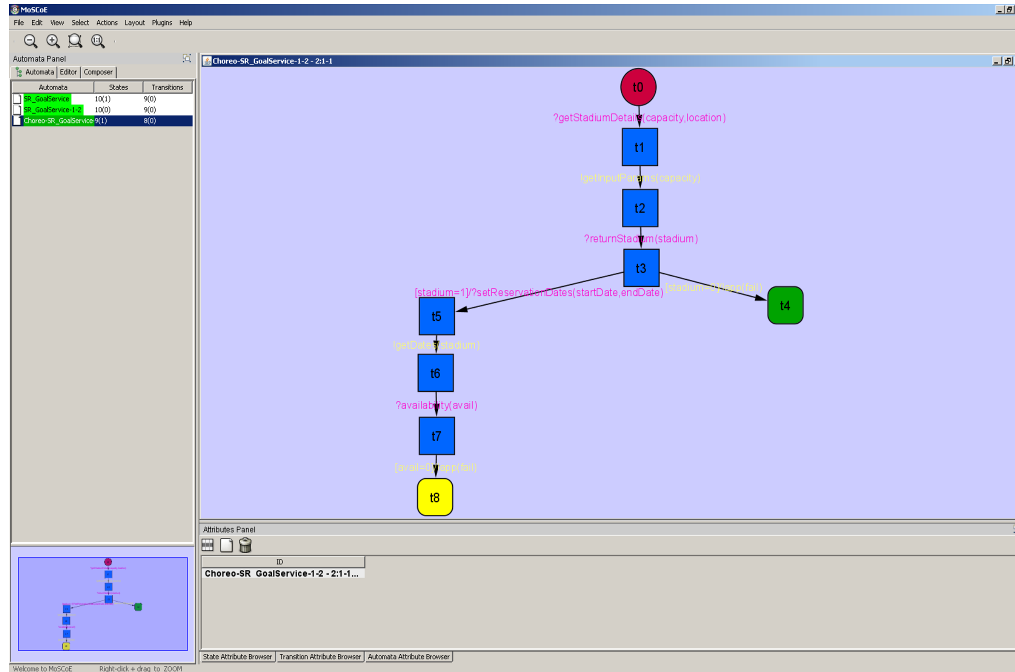

Figure 7.13 Stadium Reservation System: Choreographer corresponding to suggested goal service

Note: The current implementation does not simulate the suggested goal services for a failure in parallel. The suggested goal services are run in sequential order with the help of queue. The composer objects are placed in the queue as they are created during failure recovery. The composition algorithm is then called on each composer object as it is removed from the queue.

CHAPTER 8. CONCLUSION AND FUTURE WORK

8.1 Contribution and Future Work

The focus of this thesis is the analysis of failures that can occur in web service compositions and the identification of possible recovery options from those failures. The composition algorithm used in this work is based on the composition algorithm used in MoSCoE [3]. The composition algorithm has been modified, to facilitate the implementation of the failure analysis and recovery approach proposed in this work. This thesis also addresses the issue of what is to be presented to the user as feedback from the composition process and how that feedback is to be presented.

The specific contributions of this thesis include:

- **Modified Service Composition Algorithm based on MoSCoE composition algorithm** The composition algorithm used in this thesis is based on the composition algorithm proposed in MoSCoE [2, 3]. The changes were necessary to support the failure analysis framework proposed in this work.

- **Analysis of Composition Failures** This thesis identifies five scenarios in which failures can occur and provides techniques to identify possible recovery options for failures in each of those scenarios. For each recovery option, the goal service is automatically modified in an appropriate manner, so as to recover from that failure.

- **Resolution of Failures and Composition simulation** This work proposes that the goal service be modified based on the possible recovery solutions to a composition failure. Furthermore, the technique simulates the composition process on the modified goal service so that future failures can be identified and addressed.

- **Feedback to the user** This approach suggests that the user be given feedback in the form of a modified goal service and the choreographer corresponding to the modified goal service. In addition to these services, the feedback also contains a list of the failures that occurred and the status of the composition.

In terms of future work, we list several possible research directions below:

- **Non-functional requirements**

  The approach addressed here tackles web service composition based only on the functional requirements. The STSs that represent the goal and the component services currently support only functional requirements. Non-functional requirements are only used to select one of the possible goal services when there are multiple goal services that resolve a failure. They are expressed independently from the component services and the goal service. Future work can involve the extension of the STSs and the composition and failure analysis algorithm to handle non-functional requirements. The consideration of non-functional requirements has the benefit of reducing the number of candidate compositions that need to be considered during the composition process. Furthermore, non-functional requirements can aid in the selection of a suggested goal service to be presented to the user, when multiple equivalent goal services exist.

- **Run-time Failure Analysis**

  The composition approach discussed in this thesis applies at the design level. The composition can also fail during run-time, while the composite service is being executed. Failures could occur due to some component service being unavailable during runtime, changes in the design of the component services, or changes to the non-functional aspects of the component services. The failure analysis approach can be extended to include identification and recovery from run time composition failures.

- **Composite Functions in the Goal Service**

  The functions or atomic actions specified in the goal service might not be provided by the component services. But it is also possible that the component services provide a number

of functions which when composed together or called in a specific sequence, provide for the missing atomic action. The identification of such functions and the determination of the sequence in which they should be called can be based on the input and output parameters of the candidate functions. Thus, the failure analysis of failures caused by missing functions can be extended by attempting to find a set of functions that when called in sequence, provide the missing function.

- **Selection of optimum component service to provide the required function or input variable**

  The composition algorithm has to select a component to which to send a message in order to invoke a function or get required messages. Currently, the composition algorithm selects the first component service that it encounters in such scenarios. The algorithm can be extended to identify all the component services which provide the required function or message, and then select the best component based on certain criteria, such as Non-functional requirements.

8.2 Conclusion

One benefit of Web Service Composition is that it allows the developer to derive a very specific functionality from a set of independently developed components, in an economical manner. MoSCoe [2, 3] suggests an iterative orchestration based composition technique. During each iteration, the developer is provided feedback about the composition process and based on this feedback; the developer can reformulate the goal service. This allows the developer to start off with an abstract and possibly incomplete specification of the goal service, which can be refined in subsequent iterations. Furthermore, feedback given to the developer can help the user to understand why a composition might have failed and how to recover from this failure. This feedback can provide the developer with improved information regarding the functionalities provided by the component services.

The failure analysis and recovery technique proposed in this work addresses the issue of analyzing the failures and providing feedback to the developer. The approach identifies the

possible cause of failure and recommends that the goal service be modified if possible, for every possible recovery solution to the failure. The composition process is then simulated on each modified goal service to identify future failures. The modified goal service and the suggested choreographer are then presented to the developer as feedback, along with a list of failure messages for failures encountered during the composition process.

One form of feedback would be to identify the cause of failure and suggest to the developer a way in which the goal service can be modified to recover from this failure. The approach suggested in this work takes this a step further and modifies the goal service automatically based on the recovery option. It modifies the goal services differently for every possible way to recover from this failure. It then simulates the composition algorithm on the modified goal services to search for future failures. This approach thus examines all the possible ways in which a failure can be avoided and saves the developer from having to modify the goal service at the end of every iteration. Also, if the developer decides to adopt the modified goal service provided by the algorithm, the choreographer corresponding to the goal service is already available and the developer need not compose the modified goal service again. This approach saves the developer from experimenting with different modifications to the goal service, reducing the effort required in subsequent iterations of the composition process.

APPENDIX A. COMPOSITION ALGORITHM

Appendix A contains the composition algorithm. This algorithm is based on the composition algorithm presented in MoSCoE[3, 4]. The algorithm contains modifications required to facilitate failure analysis. These modifications include additional checks on guard conditions and changes in the handling of function invocations.

```
/**
  * r is the current stat of the goal state
  * s_i is the current state of the component service T_i
  * t is generated choreographer state
  * G is the conjunction of guard conditions
  * R is the choreographer store
  * R_i is the message store for component T_i
  */
proc GENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G) {
  if (!visited(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G)) {
    mark as visited(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G);
  } else {
    return;
  }
  forall (r --g, a--> r') && (G ∧ g) do {
    if (g is not empty && (!(G ∧ g))) { /* ∀ v ∈ g, ∃ v ∉ R */
       if (∀ v ∈ g such that v ∉ R, ∃ a T_i such that v ∈ R_i) {
          for (every such v) {
             if (v ∉ R) {
                SUBGENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G, T_i);
             }
          }
       } else {
          | Failure scenario 1: Variables required for guard condition are missing |
       }
    } else {
       case 1: a = ?m(x⃗)   /* Input action from the client */
          create transition t --g,a--> t';
          R = R ∪ x⃗;
          GENERATE(r', [s_1,s_2,...,s_n], [R_1,R_2,...,R_n], t', G ∧ g);
       case 2: a = !m(x⃗)   /* Output action to the client */
          if (x⃗ ∈ R) {
```

```
            create transition t --g,a--> t';
            GENERATE(r', [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t', G ∧ g);
         } else if (∃i such that x⃗ ∈ R_i) {
            SUBGENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G, T_i);
         } else {
            | Failure scenario 2: Variables required for output action are missing |
         }
      case 3a: a = funcName(I; O) && I ∉ R
         /* Current action is an atomic action and not all variables required to call this function
          * are available. Check the component stores for the presence of these variables
          */
         if (∀ v ∈ g such that v ∉ R, ∃ a T_i such that v ∈ R_i) {
            for (every such v) {
               if (v ∉ R) {
                  SUBGENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G, T_i);
               }
            }
         } else {
            | Failure scenario 1: Variables required for guard condition are missing |
         }
      case 3b: a = funcName(I; O) && I ∈ R && no s_i has a transition on a
         select component service T_i that houses this function;
         if (no such T_i exists) {
            | Failure scenario 4: Function required by the goal service is not available |
         } else {
            SUBGENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G, T_i);
         }
      case 3c: a = funcName(I; O) && I ∈ R && current state s_i in component T_i
                  has a transition on a
         /* Let current transition on T_i be s_i --g_i, a_i--> s'_i */
         if (G ∧ g ⇒ g_i) {
            | Failure scenario 1: Variables required for guard condition are missing |
         } else {
            done = done ∪ funcName(I; O);
            R_i = vars(a_i);
            GENERATE(r', [s_1, s_2, ..., s'_i, ..., s_n], [R_1, R_2, ..., R_n], t, G);
         }
     }
   }
}

proc SUBGENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G, T_i) {
   /* Let current transition on T_i be s_i --g_i, a_i--> s'_i */
   /* flowStack maintains an account of flowlink traversal */
  if ((a_i = !m(x⃗) || (a_i = ?m(x⃗) && x⃗ ∈ R)) && G ⇒ g_i) {
     create transition t --g_i,a_i--> t';
     R = R ∪ x⃗;
     R_i = R_i ∪ x⃗;
     GENERATE(r, [s_1, s_2, ..., s'_i, ..., s_n], [R_1, R_2, ..., R_n], t', G);
  } else if (a_i = ?m(x⃗) && x⃗ ∉ R && G ⇒ g_i) {
     if (m ∈ FL_ij) {
```

```
            msgH = m;
            k = i;
            pushIntoFlowStack((msgH, k));
        }
        while (flowStack is not empty) {
            (msgH,k) = pop(flowStack);
            while ($s_k \xrightarrow{g_k,\ a_k} s'_k$ && $header(a_k) \neq msgH$) {
                if ($a_k = ?m(\vec{y})$ && $\vec{y} \notin R$ && $G \Rightarrow g_i$) {
                    if ($m_k \in FL_{kl}$) {
                        pushIntoFlowStack((msgH, k));
                        pushIntoFlowStack(($m_k$, l));
                        continue; // to beginning of outer while loop
                    }
                } else if (($a_k = !m(\vec{y})$ || ($a_k = ?m(\vec{y})$ && $\vec{y} \in R$)) && $G \Rightarrow g_k$) {
                    create transition $t \xrightarrow{g_k,\overline{a_k}} t'$;
                    $R = R \cup vars(a_k)$;
                    $R_k = R_k \cup vars(a_k)$;
                    $GENERATE(r,\ [s_1,\ s_2, \ldots, s'_k, \ldots,\ s_n],\ [R_1,\ R_2, \ldots,\ R_n],\ t',\ G)$;
                    if ($t'$ is the root of a partial choreographer) {
                        select next transition from $s_k$;
                    }
                } else if ($a_k$ = funcName($I_k$; $O_k$) && $G \Rightarrow g_k$) {
                    if ($I_k \notin R$) {
                        [Failure scenario 3: Input variables required for function are not available]
                    } else {
                        done = done $\cup$ $a_k$;
                        $R_k = R_k \cup vars(a_k)$;
                    }
                } else {
                    [Failure scenario 1: Variables required for guard condition are missing]
                }
            }
            if ($s_k \xrightarrow{g_k,\ a_k} s'_k$ && header($a_k$) = msgH) {
                if ($G \Rightarrow g_k$) {
                    create transition $t \xrightarrow{g_k,\overline{a_k}} t'$;
                    $R = R \cup vars(a_k)$;
                    $R_k = R_k \cup vars(a_k)$;
                    $GENERATE(r,\ [s_1,\ s_2, \ldots, s'_k, \ldots,\ s_n],\ [R_1,\ R_2, \ldots,\ R_n],\ t',\ G)$;
                } else {
                    [Failure scenario 1: Variables required for guard condition are missing]
                }
            }
        }
    } else if ($a_i$ = funcName(I; O) && $I \in R$ && $G \Rightarrow g_i$) {
        done = done $\cup$ funcName(I; O); $R_i$ = $R_i \cup vars(a_i)$;
        $GENERATE(r,\ [s_1,\ s_2, \ldots, s'_i, \ldots,\ s_n],\ [R_1,\ R_2, \ldots,\ R_n],\ t,\ G)$;
    } else {
        [Failure scenario 1: Variables required for guard condition are missing]
    }
}
```

APPENDIX B. FAILURE ANALYSIS

Appendix B contains the code snippets for the different failure scenarios. Each code snippet or function describes the steps taken towards analysis and recovery for one failure scenario. Each function desrcibes the identification of the possible solutions for the scenario that it describes; as well as the suggestions made to the goal service to avoid those failures.

Failure Scenario 1: Variables required for guard condition are unavailable

```
/*
 * This function addresses failure scenario 1: Variables required for a guard
 * condition are not available. The argument to this function is the transition
 * having the guard conditions for which the variables are missing.
 */
proc missingGuardVariables(s --g, a--> s') {
  if (not the third failure in this branch of computation
    && for every v ∈ g such that v ∉ R ∃ component T_i with a transition on
    a function call f_i with v ∈ ovars(f_i)) {
      place goal service into goalServices queue;
      for (every v ∈ g) {
           - find the number of component services T_i that provide a function f_i with output
             value = v;
           - remove all the goal services from the queue and maintain the list of goal
             services temporarily
           - for (each component T_i) {
                replicate every goal service from the temporary list;
                call suggestTransitions(R', t_s, lhsVars(g), {}, s --g, a--> s', T_i); on the
                replicated goal service;
                place suggested goal service into queue;
             }
      }
      if (goalServices queue is not empty) {
           for (every goal service from queue) {
                spawn a new child thread, with copy of goal service;
                call GENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G) on
                this child thread in parallel with other child threads spawned in
                this code block;
```

```
            }
        }
    }
    manageThreadReturn(currentThreadID);
}
```

Failure Scenario 2: Variable required for output action is not unavailable

```
/*
 * This function addresses failure scenario 2: Variable required for an output
 * action condition is not available. The argument to this function is the
 * transition on the output action for which the variables are missing.
 */
proc missingOutputVariable(s --g, a--> s') {
  if (not the third failure in this branch of computation
  && for every v ∈ ovars(a) such that v ∉ R ∃ component T_i with a transition on
  a function call f_i with v ∈ ovars(f_i)) {
      place goal service into goalServices queue;
      for (every v ∈ g) {
           - find the number of component services T_i that provide a function f_i with output
             value = v;
           - remove all the goal services from the queue and maintain the list of goal
             services temporarily
           - for (each component T_i) {
                replicate every goal service from the temporary list;
                call suggestTransitions(R', t_s, ovars(a), {}, s --g, a--> s', T_i);
                replicated goal service;
                place suggested goal service into queue;
             }
      }
      if (goalServices queue is not empty) {
           for (every goal service from queue) {
                spawn a new child thread, with copy of goal service;
                call GENERATE(r, [s_1, s_2, ..., s_n], [R_1, R_2, ..., R_n], t, G) on
                this child thread in parallel with other child threads spawned in
                this code block;
           }
      }
  }
  manageThreadReturn(currentThreadID);
}
```

Failure Scenario 3: Input parameters for an atomic action are not available

```
/*
 * This function addresses failure scenario 3: Input parameters for an atomic action
 * are not available. The argument to this function is the transition on the atomic
 * action for which the input variables are missing.
```

```
 */
proc missingInputVariables($s \xrightarrow{g,\ a} s'$) {
  if (not the third failure in this branch of computation
    && for every $v \in ivars(a)$ such that $v \notin R$ $\exists$ component $T_i$ with a transition on
    a function call $f_i$ with $v \in ovars(f_i)$) {
      place goal service into goalServices queue;
      for (every $v \in ivars(a)$) {
           - find the number of component services $T_i$ that provide a function $f_i$ with output
             value = $v$;
           - remove all the goal services from the queue and maintain the list of goal
             services temporarily
           - for (each component $T_i$) {
                replicate every goal service from the temporary list;
                call $suggestTransitions(R', t_s, ivars(a), ovars(a), s \xrightarrow{g,\ a} s', T_i)$;
                replicated goal service;
                place suggested goal service into queue;
             }
      }
      if (goalServices queue is not empty) {
           for (every goal service from queue) {
                spawn a new child thread, with copy of goal service;
                call $GENERATE(r, [s_1, s_2, \ldots, s_n], [R_1, R_2, \ldots, R_n], t, G)$ on
                this child thread in parallel with other child threads spawned in
                this code block;
           }
      }
  }
  manageThreadReturn(currentThreadID);
}
```

Failure Scenario 4: Atomic action specified in the goal service is not available

```
/*
 * This function addresses failure scenario 4:a function invocation in the goal
 * service cannot be provided by the component services. The argument to this
 * function is the transition on the missing function in the goal service,
 * where $a$ = funcName(I; O)
 */
proc missingFunctionInvocation($s \xrightarrow{g,\ a} s'$) {
   if (not the third failure in this branch of computation
       && $\exists$ components with transition on a function $f$ with $ivars(f) = I$
       && $ovars(f) = O$) {
       for (every $T_i$ that provides a transition on such a function $f$) {
             /* Let the transition on the component $T_i$ be $c \xrightarrow{g_i,\ a_i} c'$ */
             - Spawn a new child thread with a copy of the goal service of this thread at
               its start state $r$, an new choreographer at start state $t$, empty component
               repositories $[R_1, \ldots, R_n]$ and empty R and with all component services
               at their start states $[s_1, \ldots, s_n]$;
             - Let $G$ represent the guard conditions, and set it to empty for this
               child thread;
```

```
            - Replace transition s --g, a--> s′ in the child thread's goal service
              with a transition on s --gi, ai--> s′
            - call GENERATE(r, [s1, s2, ..., sn], [R1, R2, ..., Rn], t, G) on
              this child thread in parallel with other child threads spawned in this
              code block;
        }
    }
    manageThreadReturn(currentThreadID);
}
```

Failure Scenario 5: Required message header cannot be provided by flow links

```
/*
 * This function addresses failure scenario 5: required message header cannot
 * be provided by flow links. The first argument to this function is the transition
 * in the goal service at which the algorithm started to explore the flow links. The
 * second argument to this function is the transition in the component service that
 * was being traversed, but for which the input message was not available
 */
proc flowLinksInsufficient(s --g, a--> s′) {
  if (not the third failure in this branch of computation
      && for every v ∈ ivars(a_k) ∃ components T_i with transition on
           a function f_i with v ∈ ovars(f_i){
      place goal service into goalServices queue;
      for (every v ∈ ivars(a_k)) {
           - find the number of component services T_i that provide a function f_i with output
             value = v;
           - remove all the goal services from the queue and maintain the list of goal
             services temporarily
           - for (each component T_i) {
                replicate every goal service from the temporary list;
                call suggestTransitions(R′, t_s, ivars(a_k), ovars(a_k), s --g, a--> s′, T_i);
                replicated goal service;
                place suggested goal service into queue;
             }
      }
      if (goalServices queue is not empty) {
           for (every goal service from queue) {
                spawn a new child thread, with copy of goal service;
                call GENERATE(r, [s1, s2, ..., sn], [R1, R2, ..., Rn], t, G) on
                this child thread in parallel with other child threads spawned in
                this code block;
           }
      }
  }
  manageThreadReturn(currentThreadID);
}
```

proc suggestTransitions

```
proc suggestTransitions(R', t_s, s --g, a--> s', T_i, s_i --g_i, a_i--> s'_i) {
  Let T_R be an ordered set of transitions. This is initially empty;
  if (ivars(a_i) ∈ R') {
     T_R = T_R ∪ {(t_s --g_i, a_i--> t_s')}
  } else {
     for (every (y ∈ ivars(a_i) such that y ∉ R') {
       recFinder(y, y, T_r);
     }
  }
  if (T_R ∩ (a transition to/from fail state s_fail) ≠ φ) {
     return partial choreographer and partial suggested goal service;
  } else {
     for suggested goal service, create chain of transitions joining
     all transitions from T_R in order of insertion;
     add transition t_s --g, a--> s' to the transition chain;
  }
}
```

proc recursiveFinder

```
proc recursiveFinder(a, b, T_R) {
  find component T_k such that T_k contains transition s_k --g_k, a_k--> s'_k
  where a_k is an atomic action and ovars(a_k) = a
  if (ivars(a_k) ∈ R') {
     Let t_s is the end state of last transition in T_R
     T_R = T_R ∪ {(t_s --g_k, a_k--> t_s')}
     R' = R' ∪ a;
  } else {
     for (every m ∈ ivars(a_k) such that m ∉ R') {
       if (m == b) {
          T_R = T_R ∪ {(s_fail --> t_s)}
       }
     } else {
       recursiveFinder(m, b, T_R);
     }
  }
}
```

Manage thread return

```
proc manageThreadReturn(threadID) {
  if (this thread is the main thread) {
     if (there exist children threads which have not been marked as failed) {
          - select one of the threads based on requested criteria and display suggested
           goal service and choreographer;
     } else { /* all the spawned threads have been marked as failed */
          - Select one thread from among the failed threads and return the corresponding
           goal service and choreographer;
```

```
      }
   }
}
```

BIBLIOGRAPHY

[1] Web Services Architecture, World Wide Web Consortium Working Group. Last accessed: 24th September, 2008. URL http://www.w3.org/TR/ws-arch/whatis

[2] J. Pathak. Interactive and verifiable web services composition, specification reformulation and substitution. PhD thesis, Iowa State University, 2007.

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

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

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

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

[7] D. Berardi, D, Calvanese, G. D. Giacomo, M. Lenzerini, M. Mecella. Automatic Service Composition based on Behavioral descriptions. *International Journal on Coop-erative Information Systems, 14(4):333-376*, 2005.

[8] Web Services Description Language (WSDL) 1.1. Last accessed: 24th September, 2008. URL http://www.w3.org/TR/wsdl.

[9] J. Rao, X Su. A survey of Automated Web Service Composition methods. *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[10] M. Pistore, P. Traverso, P. Bertoli, A. Marconi. Automated Synthesis of Executable Web Service Compositions from BPEL4WS Processes. *In 14th international conference on World Wide Web*, 2005.

[11] U. Dal Lago, M. Pistore, P. Traverso. Planning with a Language for Extended Goals. *In Proc. AAAI'02*, 2002.

[12] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Sheng. Quality Driven Web Services Composition. *Proceedings of the 12th international conference on World Wide Web*, 2003.

[13] M. Carman, L. Serafini, P.Traverso. Web Service Composition as Planning. *In The 13th International Conference on Automated Planning and Scheduling*, 2003.

[14] D. Berardi. Automatic Service Composition: Models, Techniques and Tools. PhD thesis, Universit'a di Roma, La Sapienza, Italy, 2005.

[15] J. Pathak, S. Basu, V. Honavar. Assembling Composite Web Services from Autonomous Components. *In Emerging Artificial Intelligence Applications in Computer Engineering, Frontiers in Artificial Intelligences and Applications*, 2007.

[16] J. Pathak, S. Basu, V. Honavar. Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements. *In proceedings of the 4th International Conference on Service Oriented Computing*, 2006.