Failure Analysis for Composition of Web Services Represented as Labeled Transition Systems*

Dinanath Nadkarni, Samik Basu, Vasant Honavar, and Robyn Lutz

Department of Computer Science, Iowa State University, Ames, IA 50011, USA
{yogesh, sbasu, honavar, rlutz}@cs.iastate.edu

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. Several methods have been proposed and developed to address this problem. In this paper, we consider those scenarios where the composition process may fail due to incomplete specification of goal service requirements or due to the fact that the user is unaware of the functionality provided by the existing component services. In such cases, it is desirable to have a composition algorithm that can provide feedback to the user regarding the cause of failure in the composition process. Such feedback will help guide the user to re-formulate the goal service and iterate the composition process. We propose a failure analysis technique for composition algorithms that views Web service behavior as multiple sequences of input/output events. Our technique identifies possible cause of composition failure and suggests possible recovery options to the user. We discuss our technique using a simple e-Library Web service in the context of the MoSCoE Web service composition framework.

1 Introduction

A number of formal approaches [DS05,HS04,tBBG07] have been developed in the recent years to address the problem of service composition. These approaches take as input the specification of existing service functionalities and the desired functionality (also referred to as the *goal*) in a specific formalism, and automatically generate a choreographer that mediates the communication between a subset of existing services to realize the goal (if possible). In addition to automation, these approaches also provide formal guarantees of the correctness of the composition.

Typically, the existing approaches can be viewed as a single-step process, where the result is either a feasible composite service or no result at all when the composition process fails to generate a composite service that conforms to the goal functionality. We claim that such failures may be due to the fact that

* This work is supported in part by NSF grant CCF0702758.

the developer may not be aware of all the details of existing services' functionalities and as a result s/he may specify certain goal functionality that is impossible to realize using any of the existing services. However, if the developer were provided with some feedback and/or suggestions regarding the cause of composition failure, then the developer would be able to reformulate the goal functionality without violating the overall desired requirements such that the new goal would become realizable from the composition of existing services. Such a process may be iterative resulting from multiple composition failures, failure analysis and re-formulations.

In this context, we propose methods to analyze the cause of composition failures and to provide feedback to the developers based on the analysis. We consider the problem in the MoSCoE service composition framework [PBLH06], where services and the goal functionalities are described as labeled transition systems. States in the transition system represent the configurations of the service/goal and transitions labeled with input/output events represent how the service evolves from one configuration to another. The composition algorithm in MoSCoE aims to identify the communication pattern between existing services via a choreographer such that the resulting transition system describing the composite service mimics every behavior of the goal service. Failure to generate a composite service in MoSCoE, therefore, is due to the existence of transitions in the goal that cannot be replicated by any composition of the existing services. This, in turn, implies that the given input sequence as specified by the goal functionality is not sufficient to produce the required output sequences. Once our method identifies the cause of the failure, it suggests possible changes to the goal transition system that can address the failure and lead to a successful composition. The developer can then choose to incorporate the suggestions and re-run the composition process.

The rest of the paper is organized as follows. Section 2 presents an illustrative example that will be used in the rest of the paper to explain the salient aspects of our work. Section 3 provides a brief overview of the MoSCoE composition algorithm. Section 4 discusses the various scenarios that can cause the failure of the composition followed by our method to identify them. Section 5 discusses the application of our method on the illustrative example. Section 6 gives a summary of our work and describes future avenues of research.

2 Illustrative Example

Consider a library book reservation service (`eLibrary`) that requires three main functionalities: book searches, book delivery requests and book reservations. The goal of the service is to allow a library member to search through the library catalog for a book based on parameters such as book title and the author. If the library has copies of the book, the service checks if a copy is available to be checked out. If it is, the service 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 service places a hold request on

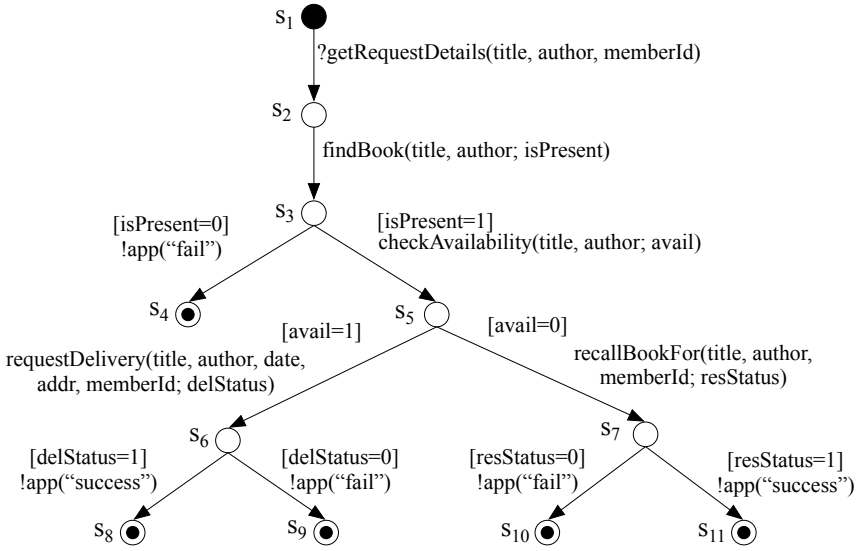
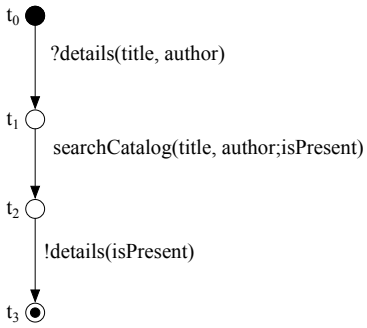


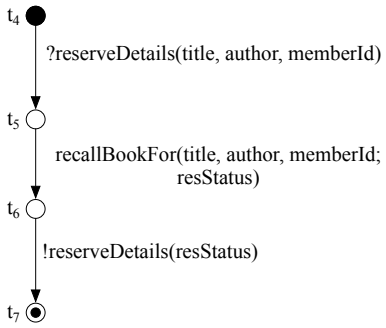
Fig. 1. Specification of goal service `eLibrary` as labeled transition system.

the book. The Web service developer is assigned the task of generating the above service. The developer prepares the transition system (Figure 1) representing the goal behavior. There are three types of transitions: input, denoted by `?`, output, denoted by `!` and function invocation. For instance $s_1 \longrightarrow s_2$ is an input transition, $s_3 \longrightarrow s_4$ is an output transition and $s_2 \longrightarrow s_3$ is a function invocation. Input/output transitions contain a message header and a message body, e.g., `getRequestDetails` is the message header and its parameters denote the message body. Transitions denoting function invocation contain the function name, the input parameters and the output of the function. For instance, `findBook` is the function name, `title` and `author` are input parameters and `isPresent` is the output. Transitions may also contain a guard (enclosed in `[·]`) denoting the condition under which the transition is enabled; transitions with no guards are always enabled.

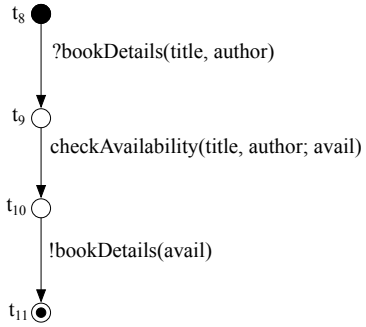
The repository that will be used to generate the composite service contains the following services: `Availability`, `BookReservation`, `DeliveryRequest`, `MemberAddress` and `SearchBook`. The `Availability` service accepts as input the title of a book and checks if a copy of the book can be checked out. The `BookReservation` service accepts as input the book title and the member ID, and places a recall request for a copy of the book for that member. `DeliveryRequest` places a request for delivery of a book to the address specified in the member's account on a particular date. `MemberAddress` accesses the member's account details and returns the member's home address. Finally, the `SearchBook` service searches the library catalog for a book given the title and the name of the



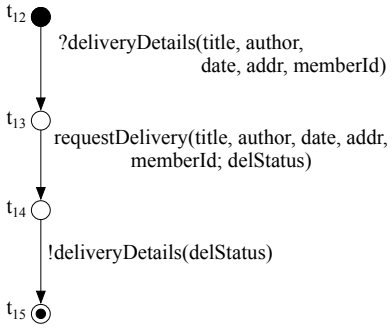
(a) `SearchBook`



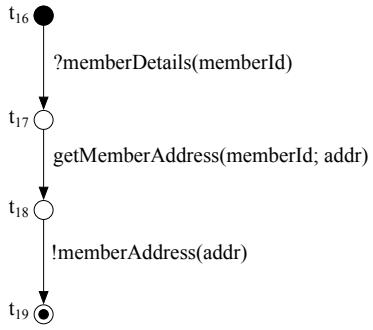
(b) `BookReservation`



(c) `Availability`



(d) `DeliveryRequest`



(e) `MemberAddress`

Fig. 2. Component services

author. The labeled transition system specification of each of these services are shown in Figure 2. The objective of the composition algorithm is to generate a choreographer that will mediate the communication between the component services such that the behavior of the component services and the choreographer replicates the behavior of the goal service. Note that the choreographer cannot generate any messages and cannot provide any functions.

We consider this simple example scenario to explain the salient aspects of the proposed method. It is worth mentioning that though the existing services in the example do not contain any loops and branching-behavior, the composition algorithm we considered (MoSCoE see Section 3) and the proposed failure analysis based on MoSCoE (see Section 4) are capable of handling services with loops and branches.

3 MoSCoE Composition Algorithm

3.1 Web services as Transition Systems

Definition 1 (Service Transition System (STS)). *A Web service transition system is a tuple $W = (S, s_0, F, T)$ where S is the set of states, $s_0 \in S$ is the start state, $F \subseteq S$ is the set of final states and T is the set of transitions between pairs of states. A transition is of the form $s \xrightarrow{g,a} s'$ where $s \in S$ and $s' \in S$ are source and destination states of the transition, g is the guard on the transition and a is the event that is executed by the transition. There are four types of events:*

- *input events denoted by* `?msgHeader(msgBody)`,
- *output events denoted by* `!msgHeader(msgBody)`, *and*
- *function invocation denoted by* `functionName(InputParameters; Output)`.
- *internal event denoted by* τ.

Transitions are referred to as input, output, function, or internal based on their labels. Internal event denotes computation or communication performed by a (composite) service that is not observable to the client.

Example 1. Figures 1 and 2 illustrate STS-representations of goal and existing services. The state states are represented by • and the final states are represented by a ⦿. In Figure 1, the transition $s_1 \longrightarrow s_2$ is an input transition with *true* guard; the transition $s_2 \longrightarrow s_3$ is a function invocation with the name of the function `findBook` and with *true* guard; the transition $s_3 \longrightarrow s_4$ is an output transition with a guard on `isPresent`.

Definition 2 (Parallel Composition of STS [PBLH06]). *Given two STSs $W_1 = (S_1, s_{01}, F_1, T_1)$ and $W_2 = (S_2, s_{02}, F_2, T_2)$, their parallel composition under the restriction set L, denoted by $(W_1 \| W_2) \backslash L$, is a tuple (S, s_0, F, T) where $S \subseteq S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$, $F \subseteq F_1 \times F_2$ and T is the transition relation described as follows: for $(i, j) \in \{(1, 2), (2, 1)\}$*

1. $s \xrightarrow{g_i, ?m(\boldsymbol{x})} s' \in T_i \ \wedge \ t \xrightarrow{g_2, !m(\boldsymbol{x})} t' \in T_j \ \wedge \ m \in L \Rightarrow (s,t) \xrightarrow{g_1 \wedge g_2, \tau} (s',t')) \in T$
2. $s \xrightarrow{g_i, e} s' \in T_i \ \wedge \ \texttt{header}(e) \notin L \Rightarrow (s,t) \xrightarrow{g_i, e} s'(s',t) \in T.$

where

$$\texttt{header}(e) = \begin{cases} m \text{ if } e \in \{?m(\boldsymbol{x}), !m(\boldsymbol{x})\} \\ \perp \text{ otherwise} \end{cases}$$

The parallel composition describes the rules by which two or more services can communicate with each other. The first rule in the transition relation describes a *synchronous* move where one services provides an output that is consumed as input by the other service. The result is an internal transition in the composite service. The second rule, on the other hand, is an autonomous move by the individual services. These rules are derived from CCS-style synchronization in process algebra [Mil82].

3.2 The Service Composition Problem

Given the transition system of a goal, W_g, and the set of available components $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$, the problem of service composition entails identifying a choreographer transition system W_c such that

$$((W_{i1} \| W_{i2} \| \ldots \| W_{ik}) \| W_c) \setminus L \approx W_g \tag{1}$$

where L is the set of actions that are not present in W_g and $\approx$ is the largest relation describing *weak bisimilarity* [Mil82] between pairs of states. The above ensures that the composite service containing $W_{i1}, \ldots, W_{ik}$ and W_c, and the goal W_g exhibit observable behaviors that are temporally indistinguishable (i.e., no temporal logic can differentiate between the behaviors). Note that, the weak bisimulation is only concerned with the observable events, i.e, all internal and unobservable behaviors (events involving τ) are ignored. The choreographer generated by the composition algorithm can be viewed as a new service with the restriction that it cannot have any function invocation or internal event; the role of the choreographer is to buffer and relay messages between existing services.

The composition algorithm takes as input the start states of the existing component service transition systems and that of the goal transition system and iteratively performs the following computations:

0. a transition in the goal is enabled only when the variables in the corresponding transition-guard are available at the current state of the choreographer;
1. if the transition in the goal is an input, then generate the corresponding input transition in the choreographer, and move the goal transition system to the next state;
2. if the transition in the goal is an output and the output messages are available to the choreographer, then generate the corresponding output transition in the choreographer, and move the goal transition system to the next state;

3. if the transition in the goal is a function and there exists a service W_i that can supply the function transition from its current state, then move W_i and the goal transition state to their corresponding next states;
4. if the transition in the goal is a function and there exists a service that can supply the function transition from some future state, then identify any service that can make a move on some input available to the current state of the choreographer and move the choreographer and the corresponding service to their next states.

Consider the illustrative example in Figures 1 and 2 (Section 2). The transition in the goal $s_1 \longrightarrow s_2$ is an input transition—this indicates that the goal service expects input from the client (who will use the service) to move from s_1 to s_2. Therefore, following Rule 1 above, the transition is replicated in the choreographer, which will act as the interface between the client and the existing services. A pair of new states c_1 and c_2 are created for the choreographer such that $c_1 \longrightarrow c_2$ is labeled by the input event that labels $s_1 \longrightarrow s_2$. At state c_2, the messages `title`, `author` and `memberId` are available to the choreographer as these are supplied by the client. If the goal is at state s_3, for the choreographer to replicate any of the transitions $s_3 \longrightarrow s_4$ and $s_3 \longrightarrow s_5$, the choreographer needs to be at a state where `isPresent` is available to the choreographer (see Rule 0 above).

If after repeated applications of the rules described above, the goal moves to the final state, then the parallel composition of the generated choreographer and the existing services is weakly bisimilar (see Equation 1 in Section 3.2) to the given goal (*soundness*). On the other hand, if none of the above rules for choreographer generation are applicable, then the composition process fails and there exists no choreographer that can realize the given goal (*completeness*). For details of the described method, and the proof of its soundness and completeness refer to [PBLH06].

The above method is described in the context of the MoSCoE service composition framework. Similar methods based in transition system representation of services and goal are developed by [BCD+05,CDL+08,HB03].

4 Failure Analysis for MoSCoE Composition

In this paper, we focus on the cases where the service composition process fails, i.e., during the iterative choreographer generation process described in Section 3 the goal, the generated choreographer and the existing services move to states from which none of the rules can be applied. We augment the composition algorithm with a method which identifies the cause of the failure of composition process and provides feedbacks/suggestions (to avoid failure) to the developer.

The feedback given to the developer is of utmost importance as it allows for progressive service composition. The feedback provides information not only regarding the cause of the composition failure but also as to how such failure can be resolved. Two problems need to be addressed for developing a feedback

process that can be effectively used in practice: (a) what is to be given as feedback and (b) how that feedback is to be presented to the developer.

4.1 Tree of Recovery Options

When failure occurs during composition, the cause of the failure is identified. Failures occur mostly due to missing messages that are required as input to functions or input transitions to some service, or due to functions required by the goal and not provided by any of the existing services. Such failures are resolved by finding alternate paths in the component services, or by calling on components that provide the missing message sets, or by identifying a semantically equivalent function in the existing services. In short, multiple resolutions are possible for each failure.

The failure analysis approach presented in this work explores each such recovery option. Upon failure, we identify all possible recovery options and a *choice point* is created in the composition computation. For each recovery option, a new branch in computation is explored using the corresponding option. 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 continues using the modified goal service. If one or more branches (at each choice point), leading to a different modification to the original goal service, eventually terminate successfully, the developer is provided with the information regarding the various goal service modifications. The developer can decide to either select one of these modifications or to discard all modifications and reformulate the goal from scratch. We refer to the above computation process as the *recovery tree*.

A sample computation tree is illustrated in Figure 3. The composition process starts and fails for the first time at node n_1. Three branches are created from this node with 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; for the purpose of efficiency we do not consider exploring any paths that contain more than three choice points. This is because the modifications deployed to the goal service in multiple choice points are likely to make the modified goal service excessively different from the original developer-specified goal service and as such are likely to be discarded by the developer.

4.2 Identifying Recovery Options

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

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

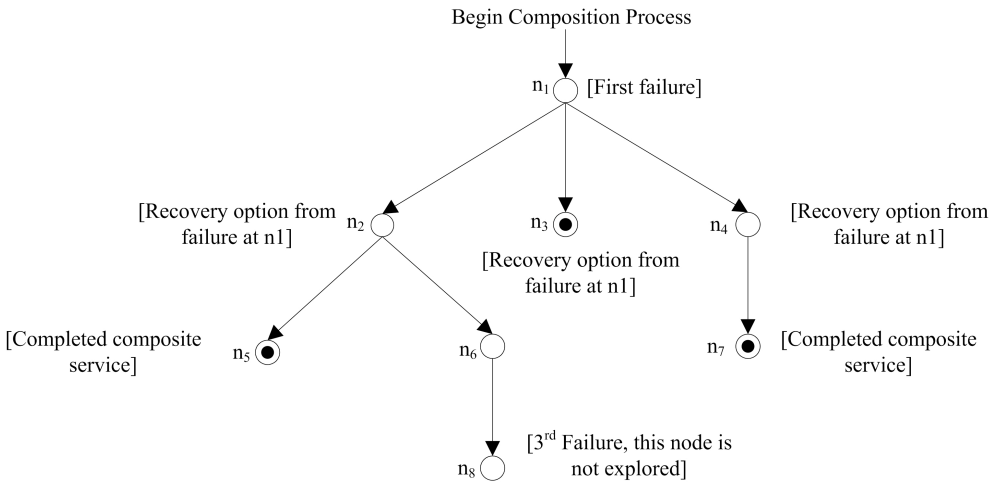


Fig. 3. Recovery Tree

2. Messages for an output transition are not available to the choreographer
3. Messages for an input transition to an existing service are not available to the choreographer
4. A required function cannot be invoked, that is, none of the existing services provide a function specified in the goal service

In the following, we discuss the recovery options identified for each of the above failure scenarios.

I: Failure due to guard conditions. This can happen when a variable (either provided via input from the client or output from some existing service) has not yet been available to the generated choreographer, and the composition process encounters a guard on such a variable in the goal transition. There are two possible recovery options from this failure based on the following:

- There exists a service that has an output containing the missing variable(s) in the output message body. The recovery option is to identify the input that must be provided at the current state of this service such that eventually the required output can be obtained.
- If no such service exists, the goal service is modified to include an input transition requesting the client to provide the message that is required for the guard condition.

II: Failure due to unavailability of output message. In this scenario, the composite service has to provide 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 transition in the goal service and the choreographer

store does not have the output message for such a transition. Recovery from this failure is based on:

– There exists a service that performs the required output action as specified by the goal. The recovery option is to identify the input that must be provided at the current state of this service such that eventually the required output can be obtained.

III: Failure due to unavailability of input message. In this scenario, the choreographer has to provide an input message to the component, which might fail if the choreographer does not have the required message set. 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 recovery option in this case is the same as scenario I above.

IV: Failure due to unavailability of required function. In this case, the goal service has a function invocation that is not provided by any of the existing services. 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 is then replaced by a transition on this semantically equivalent function.

5 Case Study

We discuss the application of the failure identification and recovery options using the illustrative example introduced in Section 2. The objective is to solve the following problem

$$\exists W_c : (\texttt{SearchBook} \| \texttt{BookReservation} \| \texttt{Availability} \| \texttt{DeliveryRequest} \| \texttt{MemberAddress}) \| W_c) \backslash L \approx \texttt{eLibrary}$$

and identify a W_c (choreographer, if one exists). In the above the restriction set $L = \{m \mid m = \texttt{header}(e) \wedge e \in \text{existing services}\}$, i.e., L contains the header of events on which the existing services can communicate with the generated choreographer (see Section 3.2).

The composition algorithm takes as input the goal as specified by the developer and the set of component services, and attempts to create a choreographer for the eLibrary system. The input action `?getRequestDetail(title, author, memberId)` in the goal corresponds to the receipt of a message from the client, meaning that the client has entered data in the system. The choreographer mimics this input action and stores the message body in the choreographer message store.

The next step in the composition is to create a transition that realizes the function invocation `findBook` (see Figure 1). The composition process fails as none of the component services can provide the required invocation (see scenario IV in Section 4.2). The failure analysis method identifies the `searchCatalog`

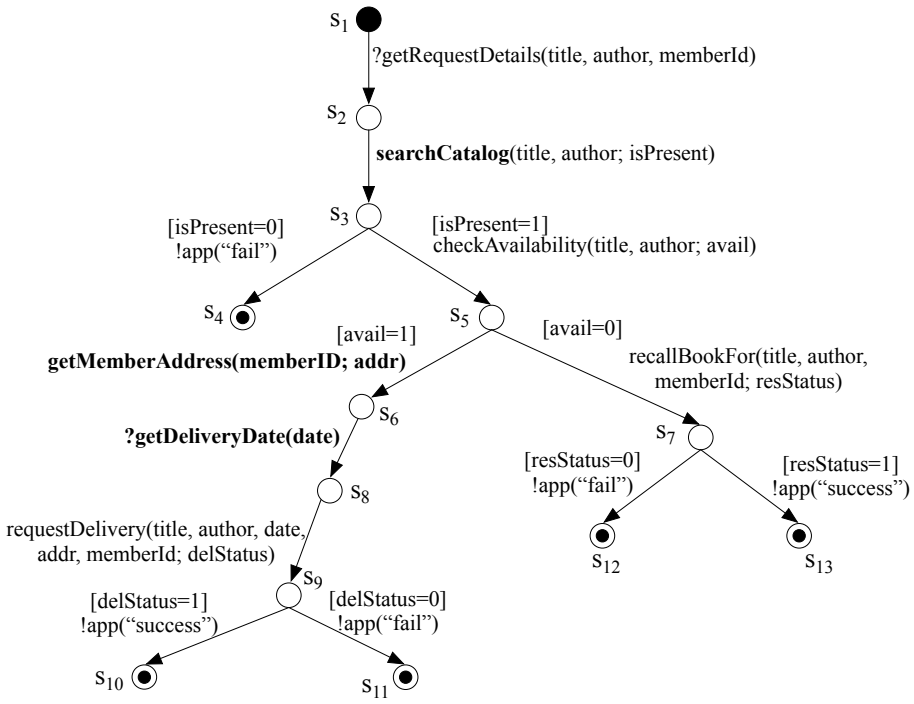


Fig. 4. Suggested Goal; modifications suggested as part of recovery from composition failure are shown in **bold**.

function provided by the `searchBook` service as a possible replacement to the findBook function, based on matching input parameters and output. The replacement function should be compared with the original for semantic equivalence. Checking a function for semantic equivalence[1] can be done based on the type of the inputs and outputs associated with the function (see for example, [Bur04,BHS03,DMD+03]). Based on this recovery option, a choice point and a branch are created in the recovery tree. The branch considers a modified version of the goal service where the function `findBook` is replaced by `searchCatalog` (see Figure 4). All the computation along this branch uses this modified goal as input.

Note that the `searchBook` service is at state t_0. In order to replicate the function invocation `searchCatalog` in the modified goal service, it is necessary to move the `searchBook` service to state t_1. In order to realize this, the choreographer is required to provide the input to `searchBook` service at state t_0. Therefore, a transition on the input action `!details(title, author)` (to be consumed by the `searchBook` service with the same message header) is created in the choreographer (see Figure 5); all the elements in the message body

[1] Methods for identifying semantically equivalent functions is beyond the scope of this work.

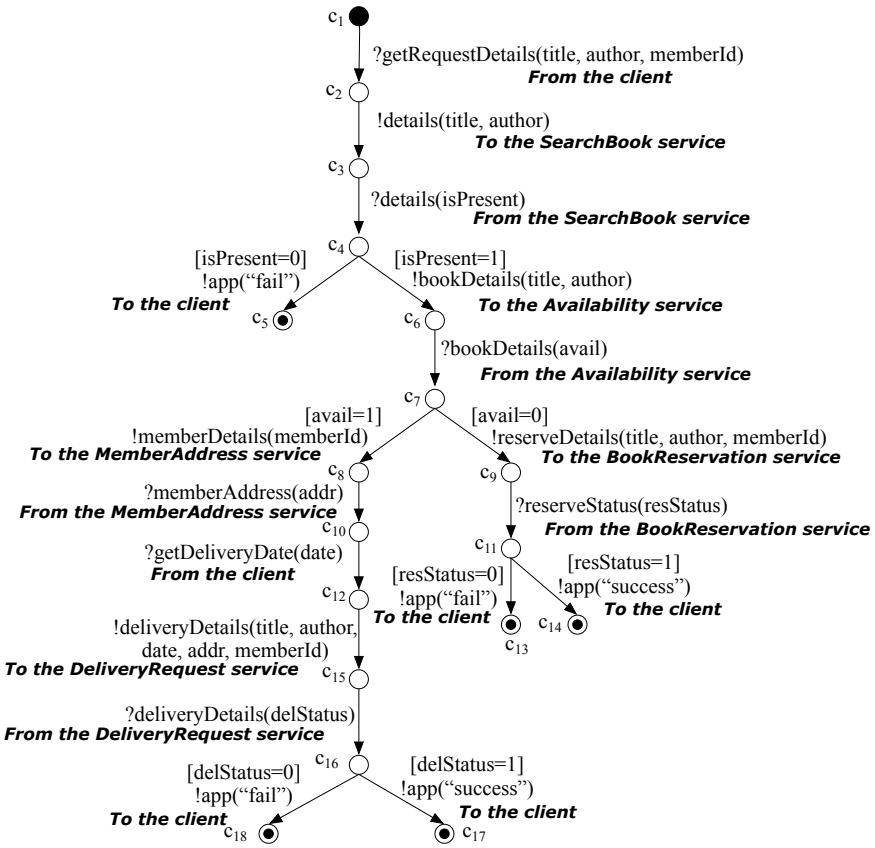


Fig. 5. Generated Choreographer for the modified Goal (Figure 4); transitions of the choreographer are annotated with the service or client (in **bold**) which participates in communication via the event on the transition.

(i.e., `title` and `author`) are available to the choreographer (i.e., present in the choreographer store).

Composition proceeds without any failure and the choreographer is generated to communicate with the `Availability` service such that the invocation of the function `checkAvailability(title, author; avail)`, as prescribed in the goal, can be realized. However, when the goal-state s_5 is reached, the composition process fails. It fails to replicate the transition $s_5 \longrightarrow s_6$. This is because the choreographer is not capable of providing `addr` information as required by the input to the function `requestDelivery(title, author, addr, memberId; delStatus)`. This corresponds to scenario III described in Section 4.2. To recover from this failure, our method searches for an existing service that has an output transition with message body containing `addr`. Such a service is identified to be `memberAddress`. A choice point and the corresponding branch of computation are created and it is given a copy of the goal service, the choreographer and the component services. The input variables required to initiate

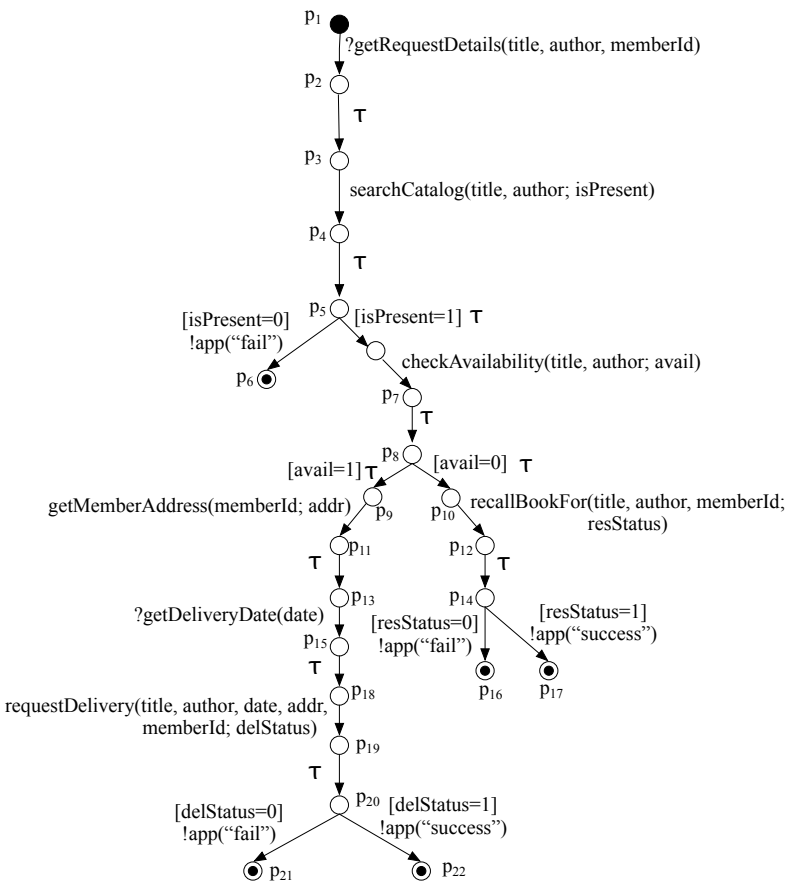


Fig. 6. Composition of generated choreographer (Figure 5) and existing services (Figure 2); composite service is weakly bisimilar to suggested goal (Figure 4).

the `memberAddress` service are available to the choreographer, so the recovery method simply inserts a transition on the `getMemberAddress` in the goal service for the new branch.

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 choreographer. At this point, the choreographer store has all the required inputs of `requestDelivery` function, except `date`. As a result, the composition process fails again.

The failure analyzer searches for any service that can provide an output `date`. As no such service exists, the only recovery option is to insert an input operation in the goal that requests the client to provide the `date`.

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(date, addr, memberId)` and `?deliveryDetails(delStatus)` in the suggested choreographer. This means

that an output message `!deliveryDetails(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 process, finally, considers the behavior of the goal as specified by transitions from $s_5 \longrightarrow s_7 \longrightarrow \ldots$. The composition process successfully completes generating the choreographer without failure.

Figure 4 shows the suggested goal following the recovery options described above; the modifications are shown in bold. Figure 5 illustrates the corresponding choreographer generated by composition process. Figure 6 shows the composite service that will be realized when the generated choreographer is composed with the existing service. It can be shown that the composite service is (weak) bisimulation equivalent to the suggested goal-transition system.

6 Conclusion

The failure analysis and the recovery techniques proposed in this work help in identifying the cause of failure in composition process and provide appropriate feedback to the developer. The feedback is described as possible modifications to the goal service for every possible recovery from the failure. Though the technique is described in the context of MoSCoE service composition framework [PBLH06], it can be applied with minimal modification for analysis of failures of composition process, where the composition is defined over different variations of labeled transition systems.

As part of future work, we plan to investigate the efficiency and applicability of the proposed method in practical settings using real-world and benchmark service composition problems. We will also work to develop failure analysis techniques where the goal is specified in the language of temporal logic (e.g., EAGLE [PTB05]), description logic ([BCG$^+$03]), etc., and the composition problem is reduced to the satisfaction problem (instead of the equivalence problem as described in MoSCoE).

References

[BCD$^+$05] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic Service Composition based on Behavioral Descriptions. *Intl. Journal on Cooperative Information Systems*, 14(4):333–376, 2005.

[BCG$^+$03] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. e-service composition by description logics based reasoning. In Diego Calvanese, Giuseppe De Giacomo, and Enrico Franconi, editors, *Description Logics*, volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[BHS03] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In *Festschrift in honor of Jörg Siekmann*, Lecture Notes in Artificial Intelligence. Springer, 2003.

[Bur04] M. Burstein. Dynamic Invocation of Semantic Web Services that use Unfamil iar Ontologies. *IEEE Intelligent Systems*, 19(4):67–73, 2004.

[CDL+08] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[DMD+03] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and Ã. Halevy. Learning to Match Ontologies on the Semantic Web. *VLDB Journal*, 12(4):303–319, 2003.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1):1–30, 2005.

[HB03] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *14th Australasian Database Conference*, pages 191–200. Australian Computer Society, Inc., 2003.

[HS04] Richard Hull and Jianwen Su. Tools for Design of Composite Web Services. In *ACM SIGMOD Intl. Conference on Management of Data*, pages 958–961, 2004.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[PBLH06] Jyotishman Pathak, Samik Basu, Robyn Lutz, and Vasant Honavar. Parallel Web Service Composition in MoSCoE: A Choreography-Based Approach. In *4th IEEE European Conference on Web Services*, pages 3–12. IEEE CS Press, 2006.

[PTB05] Marco Pistore, Paolo Traverso, and Piergiorgio Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *15th Intl. Conference on Automated Planning and Scheduling*, pages 2–11, 2005.

[tBBG07] Maurice H. ter Beek, Antonio Bucchiarone, and Stefania Gnesi. Web service composition approaches: From industrial standards to formal methods. In *ICIW*, page 15. IEEE Computer Society, 2007.