1 Web Services and Labeled Transition Systems

Web services, in essence, are software systems that can be invoked by a client over a network using standard Internet protocols to realize a desired task. Clients can take the form of a human being or another service itself, and in practice, multiple different clients might interact with the service simultaneously by executing multiple instances of the service implementation. During the interaction of the client and the service, various actions of the service will be directly executed, with the possibility of certain actions being delegated to other services. Typically, the interaction occur by the exchange of messages between the interacting parties. According to the W3C recommendation [2], a message exchange pattern (MEP) is a template that establishes a pattern for the exchange of messages between two communicating parties. A MEP identifies a common grouping of related messages. The MEPs are defined based on the client (i.e., the service requestor) and service provider, and are named based on message characteristics in the service provider, for the sake of clarity. The concept of MEPs is still evolving, and the number of patterns are potentially unlimited. However, the two basic forms of MEPs that are widely used and can be applied to construct most of the other patterns are:

- **In-only**: The client sends a message to the service provider and does not expect any related message.
- **In-Out**: The client sends a message to the service provider and expects a message.

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

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

1.1 Labeled Transition Systems

**Example 1** A client wants to search for a particular book in an online store, and if available, is willing to purchase it. Hence, it decides to interact with an available service, e-Buy, and activates an instance of the service. The service provides two functions to the client: (i) SearchBook for searching books, and (ii) PrchBook for purchasing books (if available). Assuming that the client decides to search for a particular book, it invokes SearchBook by first providing the ISBN of the book as input, and then waiting for e-Buy to finish execution of the SearchBook function. Depending on whether the requested book is available or not, a “success” or “failure” message is sent to the client. If a “failure” message is sent, the service instance terminates. On the other hand, if a “success” message is sent, the service offers the client to make a purchase by providing the quantity, shipment address and credit card information. Assuming that the client is interested in making a purchase and provides such information, e-Buy will first check whether the desired quantity (credit card) is available (valid) or not, and then execute the PrchBook function and send an appropriate message to the client depending on the success or failure of the entire operation.

Figure 1 shows the behavioral representation of the e-Buy service using a labeled transition system, which we define as follows:

**Definition 1 (Labeled Transition System [3])** A labeled transition system (LTS) is a tuple $(S, \rightarrow, s_0, S^F)$ where $S$ is a set of states represented by terms, $s_0 \in S$ is the start state, $S^F \subseteq S$ is the set of final states and $\rightarrow$ is the set of transition relations of the form $s \xrightarrow{\gamma, \alpha} t$ where:

1. an action $\alpha$ such that
   
   (a) $\text{vars}(\alpha) \subseteq \text{vars}(s)$ if $\alpha$ is an output action
   (b) $\text{vars}(\alpha) \cap \text{vars}(s) = \emptyset$ if $\alpha$ is an input action
   (c) $\text{ivars}(\alpha) \subseteq \text{vars}(s) \land \text{ovars}(\alpha) \cap \text{vars}(s) = \emptyset$ if $\alpha$ is an atomic action
2. a guard $\gamma$ such that $\text{vars}(\gamma) \subseteq \text{vars}(s)$, and
3. $\text{vars}(t) \subseteq \text{vars}(s) \cup \text{vars}(\alpha)$. 

2
Figure 1: Labeled Transition System representation of e-Buy service

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

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

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

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

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

### 1.2 Equivalence of Labeled Transition Systems

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

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

**Definition 2 (Strong Equivalence)** Given an LTS $= (S, \rightarrow, s_0, S^F)$, the strong equivalence (or bisimulation) relation with respect to substitution $\theta$, denoted by $\approx^\theta$, is a subset of $S \times S$ such that:

$$s_1 \approx^\theta s_2 \Rightarrow (\forall s_1 \theta \overset{\alpha_1}{\rightarrow} t_1 \theta : \exists s_2 \theta \overset{\alpha_2}{\rightarrow} t_2 \theta : \forall \sigma : (\alpha_1 \theta \sigma = \alpha_2 \theta \sigma) \land t_1 \approx^\theta t_2) \land s_2 \approx^\theta s_1)$$

**Definition 3 (Weak Equivalence)** Given an LTS $= (S, \rightarrow, s_0, S^F)$, the weak equivalence (or simulation) relation with respect to substitution $\theta$, denoted by $\sim^\theta$, is a subset of $S \times S$ such that:

$$s_1 \sim^\theta s_2 \Rightarrow (\forall s_1 \theta \overset{\alpha_1}{\rightarrow} t_1 \theta : \exists s_2 \theta \overset{\alpha_2}{\rightarrow} t_2 \theta : \forall \sigma : (\alpha_1 \theta \sigma = \alpha_2 \theta \sigma) \land t_1 \sim^\theta t_2)$$

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

For example, consider checking the bisimilarity of states $p_0$ and $q_0$ in the the LTSs given in Figures 2(a) & 2(b), respectively. The state $p_1$ is bisimilar to $q_1$ when $x = 0$, and is bisimilar to $q_2$ when $x \neq 0$. Similarly, $p_2$ is bisimilar to $q_1$ when $x \neq 0$, and is bisimilar to $q_2$ when $x = 0$. However, $p_0$ and $q_0$ are not bisimilar as the input action $?c(x)$ from $p_0$ to $p_1$, if matched with input action $?c(x)$ from $q_0$ to $q_1$, demands that $p_1$ and $q_1$ are bisimilar for all possible valuations of $x$ (i.e., for both $x = 0$ and $x \neq 0$). On the other hand, states $s_0$ and $r_0$ in the the LTSs given in Figures 2(c) & 2(d), respectively, are equivalent with respect to the simulation relation, since $s_0$ is simulated by $r_0$ and $s_1$ is simulated by $t_1$ for all possible valuations of $x$.

### 1.3 Composition of Labeled Transition Systems

As mentioned earlier, a client can take the form of either an human agent or another service itself. Irrespective of the client representation, the interaction between the client and the service takes place primarily by exchange of messages. For example, the client might send a message to a service requesting invocation of a particular atomic action and expect to receive to an appropriate message from the service as an outcome of the invocation. Such a conversation model between interacting services is described using the notion of a composition, which models the fact that both the interacting parties may evolve independently as a consequence of the conversation process and communicate via exchange of messages.
**Definition 4 (Composition)** Given two labeled transition systems \(LTS_1 = (S_1, \rightarrow_1, s_{01}, S_{F1})\) and \(LTS_2 = (S_2, \rightarrow_2, s_{02}, S_{F2})\), their composition, under the restriction set \(L\), is denoted by \((LTS_1 \parallel LTS_2) \setminus L = (S_{12}, \rightarrow_{12}, s_{012}, S_{F12})\) where \(S_{12} \subseteq S_1 \times S_2\), \(s_{012} = (s_{01}, s_{02})\), \(S_{F12} = \{(s_1, s_2) \mid s_1 \in S_{F1} \land s_2 \in S_{F2}\}\) and \(\rightarrow_{12}\) relation is of the form:

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

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

For example, Figure 3(c) shows the composition \(LTS_c\) of \(LTS_5\) and \(LTS_6\) (Figures 3(a) and 3(b), respectively), where \(L = \{x, y\}\). On the other hand, if \(L\) is \text{null}, i.e., the restriction set is empty, then the composition \(LTS_5\) and \(LTS_6\) can be represented by \(LTS_{c'}\) as shown in Figure 3(d).
2 Service Composition in MoSCoE

2.1 An Overview

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

Based on the definition of composition and equivalence on LTSs described in the previous section, the service composition problem can be described as:

$$\exists T_M : (\cdots ((T_M \lhd |T_i| |T_j|) \cdots |T_k|) \lhd L \approx_t T_g$$

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

2.2 Illustrative Example

Assume that a service developer is assigned to model a new Web service, Health4U, which allows senior citizens to make a doctor’s appointment to receive med-
Figure 4: LTS representation of (a) Health4U (b) The Mediator
ical attention for a particular ailment. To achieve this, Health4U relies on five existing (possibly independent) services: Appointment, MedInsurance, MedRecord, e-Ride and Validate. Appointment accepts patient data (name, ailment s/he is suffering from) and scheduling information (preferred date and time) as input to make an appointment. Appointment takes into account: (a) information about patient's insurance coverage plan to identify the designated physicians from whom the patient can receive treatment, and (b) the medical history (if any) that provides information about patient's previous appointments for the particular ailment. To obtain the needed information, Appointment communicates with MedInsurance (case (a)) and MedRecord (case (b)), both of which require the patient's SSN (Social Security Number). Appointment attempts to schedule an appointment for the patient with a physician who has treated the patient in the past. If no such physician is available, it makes an appointment with a physician who is among those designated by the insurance provider. Furthermore, Health4U arranges transportation for the patient to the medical center via the e-Ride service. This service needs the date and time for pick-up, as well as the patient's address. In addition, e-Ride communicates with Validate to determine whether the patient has provided a valid payment information (e.g., credit card) before completing the reservation.

We discuss in the next section how the composition of a service like Health4U can be accomplished by MoSCoE, which adopts the orchestrator-based model for composition. In particular, MoSCoE receives from the service developer an LTS specification of the desired goal service Health4U, as shown in Figure 4(a), which is used to construct a mediator that enables the interaction between (a subset of) the component services to provide the desired goal service functionality. Figure 4(b) shows a mediator that realizes Health4U using component services shown in Figure 5.

2.3 Algorithm for Mediator Synthesis

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

Because the task of a mediator is to orchestrate the interactions among component services, the algorithm for constructing the mediator requires information regarding dependencies between components, i.e., the dependency of an input message of a component on the output of another. For example, if a component $T_i$ requires an input of the form $?m(x)$ and a component $T_j$ provides an output of the form $!m(x)$, we say that $T_i$ is dependent on $T_j$ via the message header $m$. In such a setting, the mediator needs to synchronize with the output message from $T_j$ and pass on the output of $T_j$ as an input message to $T_i$. To make this notion of dependency more precise, we define flow links which capture the dependencies between multiple component services.
Figure 5: LTS representation of (a) Appointment (b) MedRecord (c) MedInsurance (d) e-Ride (e) Validate component services
Definition 5 (Flow Links) For services \( T_i \) and \( T_j \), if \( ?m(\vec{x}) \) and \( !m(\vec{x}) \) are present in the specifications of the respective components \( STS_i \) and \( STS_j \), then \( m \) is said to be a member of the flow link (from \( j \) to \( i \) component) set denoted by \( FL_{ij} \).

For example, consider the component services e-Ride (Figure 5(d)) and Validate (Figure 5(e)). In order for e-Ride to reserve a ride, it needs valid payment information. This information is provided by Validate after it validates the credit card information provided by the patient. Hence, there exists a flow link from Validate to e-Ride.

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

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

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

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

\[^{1}\text{In practice, there might be more than one component service that can provide the required atomic action } a, \text{ in which case, each choice is explored to find a feasible mediator.}\]
Algorithm 1 Algorithm for Modeling the Mediation & Failure-Cause Detection

/*
 * r is the goal state; s1 is the component state; t is the generated mediator state.
 * G is the conjunction of guard conditions that will be accumulated along each DFS path. All
 * variables in G are
 * universally quantified.
 * R is a store that contains all the input & output message headers of the component services.
 */

1: procedure GENERATE(r, [s1, s2, ..., sn], t, G, R)
2: if (visited(r, [s1, s2, ..., sn], t, G, R)) then // Traverse path for the first time.
3: mark as visited(r, [s1, s2, ..., sn], t, G, R);
4: end if
5: for all (r \( \Rightarrow \) r') && (G \& g)) do
6: case 1: /* input action from the client */
7: if (a = \( \tilde{r}(\bar{x}) \)) then
8: create a transition \( t \xrightarrow{a} t' \);
9: R := R \cup \bar{x};
10: call GENERATE(r', [s1, s2, ..., sn], t', G \& g, R \cup \bar{x});
11: end if
12: end case
13: case 2: /* output to the client */
14: if (a = \( \tilde{m}(\bar{x}) \)) then
15: if (\( \bar{x} \in R \)) then
16: create a transition \( t \xrightarrow{a} t' \);
17: call GENERATE(r', [s1, s2, ..., sn], t', G \& g, R);
18: else Requested output cannot be created for client. Return partial mediator;
19: end if
20: end if
21: end case
22: case 3: /* atomic action to be provided by the components */
23: if ((a = \( funcname(I; O) \)) && (no si has a transition on the action a)) then
24: select the component \( T_i \) that is capable of generating the function;
25: end if
26: if ((s, \( \tilde{r}(\bar{x}) \)) \&\& (\( \bar{x} \notin R \)) then
27: if (n \( \in FL_{1i} \)) then
28: \( msg\tilde{r} := n; k := j; \)
29: else Return partial mediator. Failure at action a.
30: end if
31: while ((s, \( \tilde{r}(\bar{x}) \)) \&\& header(a_k) \( \neq msg\tilde{r} \)) do
32: if ((a_k = \( \tilde{m}(\bar{y}) \)) \&\& (\( \bar{y} \notin R \)) then
33: if (\( a_k \in FL_{1i} \)) then
34: \( msg\tilde{r} := a_k; k := l; \)
35: end if
36: else if (((a_k = \( \tilde{m}(\bar{y}) \)) \&\& (\( \bar{y} \in R \))) then
37: if (G \& g_k) then
38: create transition \( t \xrightarrow{G \& g_k} t' \) to communicate with \( s_k \);
39: call GENERATE(r, [s1, s2, ..., s_k, ..., sn], t', G \& \bar{y});
40: if (t' is the root of a partial mediator) then
41: select next transition from \( s_k \);
42: else
43: break;
44: end if
45: end if
46: else
47: Return partial mediator. Failure at action a.
48: break;
49: end if
50: end while
if \((s_k^{g_k}, a_k) \rightarrow s'_k\) & \(header(a_k) = \text{msg}(s)\) then
if \((G \Rightarrow g_k)\) then
create transition \(t \xrightarrow{G} t'\) to communicate with \(s_k\);
call generate\(\{s_1, s_2, \ldots, s'_k, \ldots, s_n\}, t', G, R \cup \text{vars}(a_k)\);
else
Return partial mediator. Failure at action \(a\).
end if
else if \((s_k \not\in S_F^k) \lor (\text{funcName}(I; O) \not\in \text{done})\) then
Return partial mediator. Failure at action \(a\);
else
return,
end if
else if \(((s_i^{g_i}, a \rightarrow s'_i) \land (\exists x \in R) \land (G \Rightarrow g_i))\) then
create transition \(t \xrightarrow{G} t'\) to communicate with \(s_i\);
call generate\(\{s_1, s_2, \ldots, s'_i, \ldots, s_n\}, t', G, R\);
else if \(((s_i^{g_i}, a \rightarrow s'_i) \land (G \Rightarrow g_i))\) then
create transition \(t \xrightarrow{G} t'\) to communicate with \(s_i\);
call generate\(\{s_1, s_2, \ldots, s'_i, \ldots, s_n\}, t', G, R \cup \exists\);
else
Return partial mediator. Failure at action \(a\).
The last two of the preceding three scenarios are easily dealt with: the mediator transitions are generated to provide appropriate output or input message as the case may be and the procedure GENERATE is invoked recursively. Thus, in the last case, i.e., line 65, the store $R$ is updated to include the output messages from the state $s_i$. The first scenario (line 26) is more involved. As the msgSet required at the input action from state $s_i$ is not present in $R$ (line 26), the flow links (Definition 5) are explored to determine a component $T_j$ which can provide the message as output. However, it is possible that $T_j$, in turn, is at a state $s_j$ which needs a different input or output message. If the message is on input action provided by the mediator or if the message in on output action, then appropriate mediator transition is created and GENERATE is invoked recursively (lines 36--45). At line 38, $\overline{a_k}$ denotes the complement of $a_k$, i.e. $\overline{a_k} := \neg m_k(\vec{y})$ if $a_k = ?m_k(\vec{y})$; otherwise $\overline{a_k} := ?m_k(\vec{y})$. In this case, after the recursive call to GENERATE, a new transition from $s_k$ is selected at the while-condition (line 31). If the input message at $s_j$ cannot be provided by the mediator another component via flow link is selected and the process is iterated (lines 31--34).

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

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

We use a constraint solver to check the (un)-satisfiability of guards on LTS transitions. All the variables in the guard are universally quantified. At present, MoSCoE works with only equality and disequality constraints on infinite domain variables for which satisfiability checking of guards is decidable [1], although we plan to investigate a of larger classes of infinite state systems for which the construction of mediator can be made decidable [4]. The preceding algorithm may fail to construct a mediator because of either due to the absence of an action that is necessary to achieve the goal service functionality or the unsatisfiability of guards. Analysis of the cause of such failure is discussed in Section 2.4.

### 2.3.1 Modeling a Mediator for Health4U

In what follows, we show how to model a mediator for the Health4U composite service introduced in Section 2.2 using the formal framework and algorithm...
described above. Figure 4(a) shows an LTS representation of the Health4U
goal service and Figure 5 shows the corresponding LTSs of a set of available
services. Given the goal service specification and a set of available component
services, MoSCoE’s task is to construct a mediator (Figure 4(b)), which enables
the interaction between the client and component services, and is “bisimulation
equivalent” to the goal service.

The algorithm begins with the start state $s_0$ of the goal LTS and consid-
ers its transition to state $s_1$. Here, the transition takes place due to an input
action $\text{makeApp}(\ldots)$ from the client (Case 1), so MoSCoE creates an appro-
priate transition ($c_0 \rightarrow c_1$) in the mediator to receive the input message. For
the transition $s_1 \rightarrow s_2$ in the goal STS, the associated transition label is an
atomic action ($\text{SearchPhy}(\ldots)$). However, since none of the current compo-
nent states ($t_0$, $t_8$, $t_{12}$, $t_{16}$, $t_{22}$) can make a transition on this action (Case 3), the
algorithm first selects the component Appointment because it can provide the
requested function, and then creates an appropriate transition in the mediator
to send a message to Appointment. Once Appointment executes the function
$\text{SearchPhy}(\ldots)$, it transmits an output message (in this case, indicating the
availability of physician(s) for treatment of the ailment on the requested date
and time), which is received by the mediator. This behavior is modeled by
the mediator by the transition $c_2 \rightarrow c_3$ (Case 1). Depending on whether a
physician is available or not, the algorithm creates transitions $c_3 \rightarrow c_4$ and
$c_3 \rightarrow c_5$ to send/receive output/input message to/from the client (Cases 2 &
1), respectively. The algorithm proceeds in a similar fashion to model transi-
tions for atomic actions $\text{InsInfo}(\ldots)$ and $\text{PrevRec}(\ldots)$, and reach the goal
state $s_6$ and mediator state $c_9$. Now, to model a corresponding transition for
the atomic action $\text{AppPhy}(\ldots)$, the mediator refers to the message store $R$
for previous message exchanges between the client and component services, and
generates an output message $\text{inPhy}(\text{avail, elig, pre})$. Note that the values
for the variables ($\text{avail, elig, pre}$) in the message were placed in $R$ as
a result of previous message exchanges between the mediator and component
services. Since $R$ contains every message that the mediator receives from the
client and the component services, to select the relevant components (and their
messages), the mediator exploits the flow links (Definition 5) between the com-
ponents, as illustrated in Case 3 of the algorithm. This process for constructing
the mediator terminates with success when for each transition leading to a final
state in the goal, a corresponding transition in the mediator is established.

Now we proceed to discuss the scenario in which the algorithm for construct-
ing the mediator fails.

2.4 Analysis of Failure of Composition

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

1. The mediator composed with components fails to create the transition relation (see bisimulation in Definition 2). These transitions are generated by transitive closure of $\tau$-transitions obtained via synchronization between mediator and components.

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

3. The guard conditions are unsatisfiable.

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

2.4.1 Failure Cause Analysis for Health4U

In our example from Section 2.2, suppose we replace the e-Ride component service (Figure 5(d)) with component services e-Ride' and e-Ride" yielding two separate instances of the Health4U composition problem (Figure 6(a) & 6(b)). Suppose the behavior of e-Ride' is exactly the same as that of e-Ride, but it additionally requires a phone number to reserve a ride. Suppose on the other hand that e-Ride" can only reserve a ride if the time for pick-up is before 4pm. Note that in both these instances, the algorithm for constructing the mediator fails when it encounters the transition $s_9 \rightarrow s_{10}$ in the goal LTS (see Figure 4(a)). Specifically, in the case of the component service e-Ride', the actions for Health4U and e-Ride' do not match, whereas in the case of e-Ride", the corresponding guard condition is not satisfied. Thus, in the case of e-Ride' a failure results from an exception being raised either at line 59 or 69, indicating that a particular action present in the goal STS does not match with the component action for the particular transition. In the case of e-Ride" a failure arises due to an exception being raised either at line 56 or 77, indicating a mismatch in guards for the corresponding transition relation in the goal STS. MoSCoE provides such information about the cause of a failed attempt at service composition to the service developer. The developer can then reformulate the original goal specification (e.g., changing the function parameters or pre-conditions) to
realize a suitable mediator. These steps can be iterated until such a mediator is eventually realized or the user decides to abort.

2.5 Theoretical Analysis

Theorem 1 (Soundness & Completeness) Given a goal service $T_g$ with start state $s_0$ and $n$ component services $T_1 \ldots T_n$ with the corresponding start states $s_0_1 \ldots s_0_n$, the procedure $\text{generate}(s_0, [s_0_1, s_0_2, \ldots, s_0_n], t_0, \text{true}, \emptyset)$ in Algorithm 1 is guaranteed to terminate with a mediator $T_M$ with start state $t_0$ if and only if $(T_M\parallel T_1\parallel T_2) \ldots \parallel T_n) \approx_t T_g$ whenever such a mediator exists, and with a failure otherwise.

Proof Sketch: We prove the theorem by contradiction. Suppose the procedure $\text{generate}(s_0, [s_0_1, s_0_2, \ldots, s_0_n], t_0, \text{true}, \emptyset)$ in Algorithm 1 yields a mediator $T_M$ with start state $t_0$ which when used to orchestrate the component services under the restrictions imposed by the guards $L$, fails to realize the goal service $T_g$, i.e., the composition is not bisimulation equivalent to $T_g$. There are four cases to consider: (i) for an input action in $T_g$, there is no corresponding input action in $T_M$; (ii) for an output action in $T_g$, there is no corresponding output action in $T_M$; (iii) an atomic action present in $T_g$ is not modeled by the composition; and finally (iv) some sequence of actions in the goal is not provided by the composition due to the unsatisfiability of one or more guards.

However, case (i) is ruled out by the algorithm because for each message sent from the client to $T_g$, a corresponding input action is created in $T_M$ to receive the message (Case 1 of $\text{generate}$). Case (ii) is ruled out because for
each output message that is to be sent to the client (as modeled in $T_g$), a corresponding output action is created in $T_M$ if that message can be retrieved from the message store $R$ (otherwise an exception is raised resulting in termination of the algorithm with failure (Case 2 of generate)). Case (iii) is ruled out because the atomic actions in $T_g$ are modeled by first determining the component(s) that can provide the relevant functions and then creating the relevant transitions in $T_M$ to communicate with the respective component(s) (otherwise the algorithm terminates with failure). Note that the communications between $T_M$ and any $T_i$ leads to transitions labeled by $\tau$ (Definition 4). The desired goal-function will be matched by the composition after zero steps if there is a component at a state with outgoing transition labeled by the function; otherwise the composition will lead to a state with an outgoing transition labeled by the desired function, after multiple $\tau$-steps representing component-mediator synchronous communications. Finally, in all the above cases, if the guards do not match or the guards in the component(s) are stronger than those in $T_g$ (and $T_{cr}$), the algorithm terminates with an appropriate failure cause, thereby ruling out case (iv).

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

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


References


