Composing Web Services through Automatic Reformulation of Service Specifications

Jyotishman Pathak
Div. of Biomedical Informatics
Mayo Clinic College of Medicine
Rochester, MN 55905, USA
pathak.jyotishman@mayo.edu

Samik Basu
Dept. of Computer Science
Iowa State University
Ames, IA 50011, USA
sbasu@cs.iastate.edu

Vasant Honavar
Dept. of Computer Science
Iowa State University
Ames, IA 50011, USA
honavar@cs.iastate.edu

Abstract

Typical approaches to service composition seek to realize a goal service specification, described using a labeled transition system (LTS) provided by a service developer, by constructing a structurally equivalent LTS using a set of available component services (also described using LTSs) that match the input and output requirements of the transitions. As such, existing composition approaches fail to realize the goal LTS whenever available component service LTSs cannot be used to "mimic" the structure of the goal LTS. This failure requires that the service developer formulates an alternate goal LTS and re-iterates the composition step. However, the process of manual reformulation of goal LTS is both laborious and error prone. In this setting, we describe an efficient data structure and algorithms for analyzing data and control flow dependencies implicit in a user-supplied goal LTS specification to automatically generate alternate LTS specifications that capture the same overall functionality with respect to the data and control dependencies, and determine whether any of the alternatives can lead to a feasible composition. The result is a significant reduction in the need for the tedious manual intervention in reformulating LTS specifications of the goal.

1 Introduction

Service-oriented computing offers a powerful approach for the construction of complex applications from autonomously developed, distributed software components in multiple domains including e-Science, e-Business and e-Government. Consequently, there is a growing body of work on specification, verification and composition of Web services. Of particular interest are techniques for automated or semi-automated composition of a composite service that realizes user (e.g., a service developer) specified functionality (goal service) using a subset of available component services. Specifications for services can be modeled using different techniques including state charts [1], finite-state automata [2], logic programming [10], and labeled transition systems (LTS) [8, 9].

The composition algorithm typically attempts to realize the goal service by "replicating" the *structure* of the goal service specification (e.g., LTS, state chart, or similar formalism) using component services (also described using the same formalism) that match the input and output requirements of the goal. If the composition algorithm fails to realize the goal service specification, the entire process fails, thereby shifting the responsibility of identifying the cause(s) for failure of composition as well as modification of the goal specification to the service developer. In general, there can be two broad classes of scenarios in which a service composition algorithm fails to realize a goal service:

1. The desired *functionality* of the goal service *cannot* be realized by composing the available component services. In this case, the user needs to either modify the overall functionality of the desired goal service (e.g., add/delete service functions, change function parameters) and/or broaden the search for component services beyond those initially considered by the algorithm.
2. The desired *functionality* of the goal service *can* be realized by composing the available component services, but the composition algorithms fail to "mimic" the *structure* of the goal service using the available component services. In this case, it is possible to reformulate the structure of the goal service specification, without altering its overall functionality, into one that can be realized by using the existing services.

Recently, techniques have been developed for automated identification of cause(s) of failure of composition for both the scenarios [8, 9]. However, these methods suffer from the limitation that the tedious manual reformulation of an alternative goal service specification with the same functionality (case 2 above) to be tried is left to the user. We provide here a framework for *composing Web services through automatic reformulation of service specifications* to overcome this limitation.

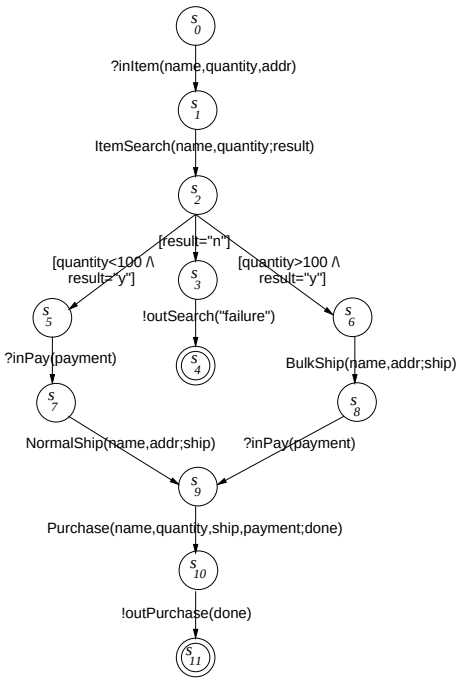
Without loss of generality, building on previous work [8, 9], we use LTS to represent the goal service provided by a service developer, the set of available component services, and the generated composite service that realizes the

goal. We show that any alternative goal LTS reformulation that does not violate the data and control dependencies that are implicit in the user-supplied goal service LTS specification is *provably functionally equivalent* to the goal service. We describe an efficient data structure, in the form of a *dependency matrix* and algorithms to maintain and analyze the data and control flow dependencies in the goal service LTS specification. This data structure is used to iteratively generate (as yet untried) alternatives that are functionally equivalent to the user-supplied goal service LTS specification until composition succeeds or no alternative reformulations remain to be tried. Because generating the complete set of alternatives that are functionally equivalent to a user-supplied goal service LTS specification is expensive and potentially wasteful, we generate the alternatives *on-the-fly*, i.e., alternatives are generated as and when needed for the verifying the existence of a functionally equivalent composition. The result is a significant reduction in the need for the tedious manual intervention in reformulating specifications by limiting such interventions to settings where both the original goal LTS as well as its alternatives cannot be realized using the available component services. The proposed method works for both orchestrator and choreographer based techniques to service composition.

2 Illustrative Example

We present a simple example where a service developer is assigned to model a new composite (or *goal*) service that allows clients to purchase items and ship them to a particular destination. To achieve this, the goal service operates as follows: (i) First, it accepts from the client as input the *name* of the item to be purchased along with the desired *quantity* and the *address* where the consignment has to be shipped. (ii) Once the input is received, it searches the particular item for the required quantity in an inventory. (iii) If the search fails, a failure message is sent to the client. But if the search succeeds, depending on the quantity of the item to be purchased, either *bulk* or *normal* shipping is checked for confirming whether the items can be shipped to the particular *address* or not. (iv) Also, if the item search succeeds, the client is asked to provide *payment* information which is eventually used for *purchasing* the items. (v) Finally, an appropriate *notification* is sent to the client indicating whether the entire process was a success or failure.

Figure 1(a) shows the representation of such a goal service, named `e-Buyer`, described using a labeled transition system. Here, `?msgHeader(msgSet)` and `!msgHeader(msgSet)` refer to the input and output actions of the service respectively. Communication between different services occurs via *synchronization* between actions with the same `msgHeader` resulting in the transfer of `msgSet` from the entity performing an output action to the one performing an input action. The service also in-



(a)

Actions	Name
a	`?inItem(name,quantity,addr)`
b	`ItemSearch(name,quantity;result)`
c	`?inPay(payment)`
d	`!outSearch("failure")`
e	`BulkShip(name,addr;ship)`
f	`NormalShip(name,addr;ship)`
g	`Purchase(name,quantity,ship,payment;done)`
h	`!outPurchase(done)`
Guards	**Name**
G_1	`[quantity<100 ∧ result="y"]`
G_2	`[quantity>100 ∧ result="y"]`
G_3	`[result="n"]`

(b)

Figure 1: (a) LTS representation of `e-Buyer` goal service (b) Index of actions and guards in `e-Buyer`

clude atomic actions denoted by `funcName(inputSet;output)`. Additionally, a transition is annotated by guards (denoted by `[guards]`) which control whether or not it is enabled; the absence of a guard implies that the guard is *true* (always enabled).

A closer analysis of Figure 1(a) reveals various data and control flow dependencies between the actions present in the `e-Buyer` service. For example, the input action `?inItem(name,quantity,addr)` has to occur *before* the atomic action `ItemSearch(name,quantity;result)` since the information required to execute the latter is provided by the former. Similarly, `NormalShip(name,addr;ship)` and `BulkShip(name,addr;ship)` are executed *mutually exclusively* depending on the valuation of the variables `quantity` and `result`.

Assume that Figures 2(a) & 2(b) presents the two available services to be used to realize `e-Buyer`. If `Search-N-Ship` is selected first, it leads to a com-

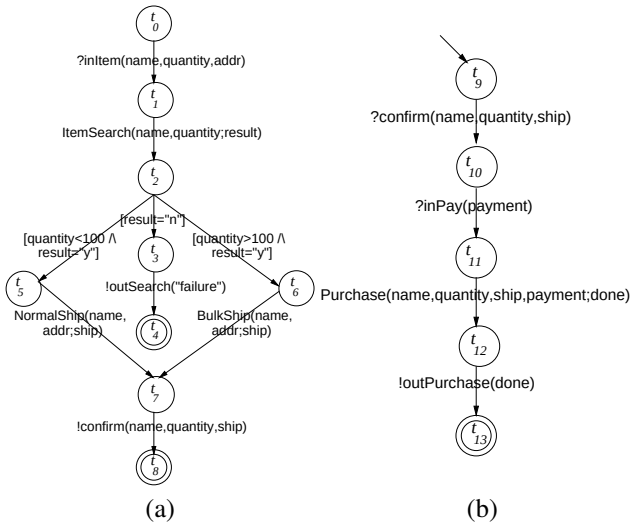


Figure 2: LTS representation of (a) `Search-N-Ship` (b) `Purchaser` component services

position failure due to different branching behaviors since `e-Buyer` requires the execution of an input action `?InPay(payment)` when [`quantity` $<$ `100` $\wedge$ `result = y`] is true, whereas `Search-N-Ship` invokes the atomic action `NormalShip` when the same condition is satisfied. Similarly, selecting the `Purchaser` service first leads to a failure as well. For both the circumstances, typically the service developer will be required to modify/reformulate the goal service representation (in this case, re-ordering transitions in the LTS representation of `e-Buyer`) and re-initiate the composition process.

Note that the above composition process failed for the original goal service `e-Buyer` since the typical composition algorithms [1, 2, 8, 9] aim to realize the *exact structural* representation of the goal service using the component services. Instead, we will show in the proceeding sections that it suffices to realize a composition using *alternate* structural representations (such as `e-Buyer'`, see Figure 3) of the original goal service, as long as the generated alternatives have the same functionality (with respect to control and data dependencies) as the original goal service.

3 Preliminaries

We represent Web services using Labeled Transition Systems (LTSs) which comprise of states denoting the configurations of a service and transitions corresponding to its evolution from one configuration to another.

Definition 1 (Labeled Transition Systems)**.** *A labeled transition system (LTS) is a tuple* $(S, \longrightarrow, s0, S^F)$ *where* S *is a set of states,* $s0 \in S$ *is the start state,* $S^F \subseteq S$ *is the set of final states and* $\longrightarrow$ *is the set of transition of the form* $s \xrightarrow{\gamma,\alpha} t$ *where* $s, t \in S$*,* γ *is the condition under which the transition is enabled and* α *is the action on the transition.*

There are three types of actions: input actions denoted

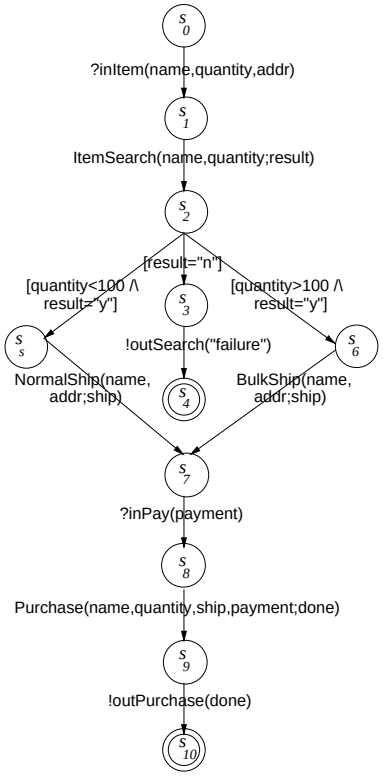


Figure 3: *Alternate* goal service LTS: `e-Buyer'`

by $?m(\overrightarrow{x})$, output actions denoted by $!m(\overrightarrow{x})$ and atomic actions denoted by $f(\overrightarrow{i}; o)$. In the above, m corresponds to message header, $\overrightarrow{x}$ corresponds to set of input/output messages, f refers to the atomic action name, and finally $\overrightarrow{i}$ and o are input and output parameters of the atomic action, respectively. We use two functions $\mathcal{I}$ and $\mathcal{O}$ to identify the input and outputs of each action and guard such that: $\mathcal{I}(?m(\overrightarrow{x})) = \emptyset$ and $\mathcal{O}(?m(\overrightarrow{x})) = \overrightarrow{x}$ since for the system executing the input action messages are produced as a result of the input action. Dually, $\mathcal{I}(!m(\overrightarrow{x})) = \overrightarrow{x}$ and $\mathcal{O}(!m(\overrightarrow{x})) = \emptyset$. The input and output variables of an atomic action are its input and output parameters, respectively. Finally, for a guard γ, $\mathcal{I}(\gamma)$ is the set of variables used in the guard while $\mathcal{O}(\gamma) = \emptyset$. Figure 1(a) shows the LTS representation of the `e-Buyer` service where $\mathcal{O}$(`?inItem(name,quantity,addr)`) = {`name,quantity,addr`}, $\mathcal{I}$(`[quantity<100` $\wedge$ `result=''y'']`) = {`quantity,result`}, and so on.

The interaction between the services takes place by exchange of messages which happens when two services synchronize over input and output actions with the same message header resulting in the transfer of messages from the output action to the input action. Such an interaction described by *synchronous composition* is defined as follows:

Definition 2 (Composition)**.** *Given* $LTS_1 = (S_1, \longrightarrow_1, s0_1, S_1^F)$ *and* $LTS_2 = (S_2, \longrightarrow_2, s0_2, S_2^F)$*, their com-*

position, under the restriction set M, is denoted by $(LTS_1 \mid\mid LTS_2)\backslash M = (S_{12}, \longrightarrow_{12}, s0_{12}, S^F_{12})$ *where* $S_{12} \subseteq S_1 \times S_2$, $s0_{12} = (s0_1, s0_2)$, $S^F_{12} = \{(s_1, s_2) \mid s_1 \in S^F_1 \wedge s_2 \in S^F_2\}$ *and* $\longrightarrow_{12}$ *relation is of the form:*

1. $s \xrightarrow{g_1, ?m(\vec{x})} s' \wedge t \xrightarrow{g_2, !m(\vec{x})} t' \wedge m \in M \Rightarrow (s,t) \xrightarrow{g_1 \wedge g_2, \tau} (s', t')$,
2. $s \xrightarrow{g_1, \alpha} s' \wedge header(\alpha) \notin M \Rightarrow (s,t) \xrightarrow{g_1, \alpha} (s', t)$, *and*
3. $t \xrightarrow{g_2, \alpha} t' \wedge header(\alpha) \notin M \Rightarrow (s,t) \xrightarrow{g_2, \alpha} (s, t')$.

Here, the restriction set M includes the message headers on which the participating LTSs must synchronize and generate a τ-action. We use $header(\alpha)$ to return the message header of input and output actions; for atomic actions and τ-actions it returns a constant which is never present in M.

Based on the definitions above, we say that if L_g represents the goal service (modeled by the service developer) and $L_1, L_2, \ldots, L_n$ is the set of available component services, then determining a feasible composition entails selecting a set of suitable component services and generating a composite service L_C such that: $\exists L_C : (\ldots((L_C||L_i)||L_j)||\ldots||L_k)\backslash M \approx_w L_g$, where M contains all the input and output message headers of the component services and $\approx_w$ denotes the *equivalence* relationship[1], essentially, ensuring that interaction between the composite service L_C and the component services conforms to the desired goal service L_g. For details of the composition algorithm based on $\approx_w$ refer to [8, 9].

4 Reformulation-based Service Composition

Realization of a goal service amounts to identifying a composite service whose LTS representation is structurally equivalent ($\approx_w$) to the goal LTS. However, a goal service can be safely said to be realized from a composite service as long as the control and data flow dependencies of the former are satisfied by the latter. We will say that two services are *functionally* equivalent if they have the same control and data flow dependencies. As satisfying structural equivalence is a stronger requirement than conforming to data and control flow dependencies (i.e., two LTSs that are structurally non-equivalent can still have same data and control flow), we will present a novel technique to automatically adapt the structure of the goal service without altering the control and data flows, such that different structurally variant but functionally invariant goal LTSs are synthesized and verified for realization from existing component services.

4.1 Functionally Equivalent Web Services

We define the notion of functional equivalence of two services in terms of the corresponding equivalence between the respective LTSs. Given an LTS $L = (S, \longrightarrow, s0, S^F)$, its behavior can be represented as the sequences of actions and guards from $s0$ to some state in S^F. Notationally, we will describe such a sequence as a string $(s_0\sigma_0)(s_1\sigma_1)\ldots(s_n\sigma_n)$, where WLOG we can assume that every transition in LTS is labeled either with an action or a guard and $\forall i (0 \leq i \leq n) : s_i \xrightarrow{\sigma_i} s_{i+1}$ such that $s_0 = s0$ and $s_{n+1} \in S^F$. We refer to all such sequences in L as the behavior of L and denote it by $\mathcal{B}(L)$.

Definition 3 (Functional Equivalence)**.** *An LTS L_1 is said to be functionally simulated by an LTS L_2, denoted by $L_1 \sqsubseteq L_2$, if and only if for all $\texttt{seq} = (s_0\sigma_0)(s_1\sigma_1)\ldots(s_n\sigma_n) \in \mathcal{B}(L_1)$ there exists $\texttt{seq}' \in \mathcal{B}(L_2)$ such that $\texttt{seq}$ and $\texttt{seq}'$ are permutation of each other with the following conditions:*

1. *For $i < j$, $(s_i\sigma_i)$ appears before $(s_j\sigma_j)$ in $\texttt{seq}'$ if $\mathcal{O}(\sigma_i) \cap \mathcal{I}(\sigma_j) \neq \emptyset$.*
2. *For $i < j$, $(s_i\sigma_i)$ appears before $(s_j\sigma_j)$ in $\texttt{seq}'$ if s_i is a branch point in L_1 and σ_j does not appear in all sequences in $\mathcal{B}(L_1)$ that contains s_i.*

L_1 and L_2 are functionally equivalent, denoted by $L_1 \equiv L_2$, if and only if $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$.

In the above, the first condition asserts that if LTS L_1 demands input of an operation (for either an action or guard) which must be obtained from the output of another operation, then it must be conformed by the corresponding sequence in LTS L_2 (*data flow dependency*). The second condition ensures that if an operation depends on a guard (i.e., appears in a specific branch) in L_1, it must similarly depend on the same guard in L_2 (*control flow dependency*). Functional equivalence demands that L_1 and L_2 *functionally simulate* each other. Note that this notion of equivalence is different from simulation or bisimulation equivalence applied traditionally in process algebra [6] where equivalences define structural similarities.

The control and data flow dependencies are succinctly captured in a novel data structure called *dependency matrix* which forms the core of our technique for reformulation-based composition. During composition, the goal LTS is explored and verified whether every sequence and branch behavior present in the LTS is realizable. In the event, there is a failure, the composition algorithm is backtracked and alternate functionally equivalent goal LTSs, obtained using dependency matrix, are explored.

Definition 4 (Dependency Matrix)**.** *Given an LTS $L = (S, s0, \longrightarrow, S^F)$, its dependency matrix D_L is a $N \times N$ matrix, where N is the number of actions (atomic, input and output) and guards in the transition-labels. For a row i and column j in D_L, the cell $C_{i,j}$ is assigned such that:*

- *if $C_{i,j} = \{X\}$, then the i-th element is control-dependent on the j-th element. The assignment X is done*

[1]The equivalence relation $\approx_w$ is referred to as observational equivalence and captures equivalent branching behavior of transition systems modulo the τ-transitions/actions [6].

between action-guard or guard-guard pairs denoting that the guard j has to be analyzed before action i can be executed or guard i can be analyzed.

- *if $C_{i,j} = \mathcal{I}(i) \cap \mathcal{O}(j)$, then the i-th element is data-dependent on the j-th element i.e., outputs from the j-th element are used for the analysis/evaluation of the i-th element. This assignment is done between guard-action or action-action pairs.*
- *if $C_{i,j} = \{Y\}$, then the i-th and j-th elements are guards which label different transitions from a branch-point in L. That is, the elements i and j are mutually exclusive and cannot appear in the same path in L.*

Figure 4(a) shows the dependency matrix of `e-Buyer` (Figure 1(a)). For example, it states that action `a` (`?inItem`) has to occur before action `b` (`ItemSearch`) because the variables `name` and `quantity` required to execute `b` (input parameters of `ItemSearch`) are provided by `a`. Similarly, the guards G_1, G_2 and G_3 are mutually exclusive since they appear in separate execution paths. Here, `a`, `b`, G_1, G_2, and G_3 correspond to actions and guards in Figure 1(b).

Theorem 1. *For any two LTSs L and L', $L \equiv L'$ if and only if their dependency matrices D_L and $D_{L'}$ are identical.* □

4.2 Generation of the Dependency Matrix

Identifying Data Flow Dependency. Procedure DATA FLOWDEP in Algorithm 1 takes as argument the current state (`curr`) of the LTS being explored for dependency analysis, the operation (guard or the action, `Op`) and the set of variables (`VSet`) in `Op` whose data dependencies are being analyzed. Backward exploration of the LTS is performed from `curr`. If a parent-state is reachable via an action "a" such that the intersection of its output and `VSet` is non-empty (`lines 6--9`), then the corresponding cell in the dependency matrix ($C_{Op,a}$) is assigned `output(a)`. The `VSet` is updated by removing `output(a)` from the set. Finally, the procedure is recursively invoked (`line 10`). The recursion terminates (`lines 2--4`) when `VSet` is empty (i.e., all the data-dependencies of `VSet` have been identified) or `curr` is the start-state of the transition system (i.e., there exists no incoming transition to `curr`). For example, invocation of DATAFLOWDEP(s_1, `ItemSearch`, {`name,quantity; result`}) in `e-Buyer` of Figure 1(a), will create a dependency with the action `?inItem` since it provides the inputs required to execute `ItemSearch`.

Identifying Control Flow Dependency. Procedure CTRL FLOWDEP in Algorithm 2 identifies the control dependencies in an LTS and is similar to DATAFLOWDEP with the exception that instead of checking for data intersection, CTRLFLOWDEP checks whether the `parent` is a branch point (`line 6`). In particular, the procedure conservatively classifies operations as control dependent on a branch-point

Algorithm 1 Identifying Data Flow Dependency

```
1: procedure DataFlowDep(curr, Op, VSet)
2:   if (curr is a start-state OR VSet = ∅) then
3:     return
4:   end if
5:   for all parent --g,a--> curr do
6:     if (output(a) ∩ VSet ≠ ∅) then
7:       C_{Op,a} := output(a)
8:       VSet := VSet − output(a)
9:     end if
10:    DataFlowDep(parent, Op, VSet)
11:  end for
12: end procedure
```

(more precisely on the guard associated with the branch-point) in which it appears. However, there are cases where an operation might appear in all the possible branches. In such a situation, the said operation is *not* control dependent on the branch-point as it is invoked for all possible valuation of the guards at the branch-point. In order to precisely identify control dependencies by eliminating such cases, CTRLFLOWDEP is followed by invocation of UPDATECTRLFLOWDEP (Algorithm 2). This procedure identifies (a) the set of paths originating from each branch-point to a state which is either a final state and/or a joint point of the branch (`line 13`), and (b) the set of guards at a branch-point (`line 14`). If there is an operation `Op` (obtained from procedure CTRLFLOWDEP) which is dependent on at least one guard associated with the branch-point (`line 15`) and also appears in *all* paths in T (`line 16`), then `Op` is *not* control-dependent on any guard appearing in the branch-point under consideration. Accordingly, all the `X`s in $C_{Op,g}$ are removed (`line 17`). Furthermore, `Y` is assigned to the cells corresponding to the guards associated with the same branch point as all the guards cannot evaluate to true simultaneously, i.e., they are mutually exclusive (`lines 20--21`). For example, invocation of UPDATECTRLFLOWDEP(s_2) in `e-Buyer` of Figure 1(a) will assign `Y`s to the cells C_{G_1,G_2}, C_{G_1,G_3}, C_{G_2,G_3}, C_{G_2,G_1}, C_{G_3,G_1} and C_{G_3,G_2}.

4.3 Algorithm for Reformulation

Our technique allows reformulation during composition, that is, given a goal LTS L which results into failure of composition, the technique automatically reformulates L using its dependency matrix to identify a functionally equivalent L' and checks for feasible compositions. L' is generated as and when the composition feasibility is checked, as opposed to, generating L' first and then checking for its feasibility. This is necessary because generating the complete set of alternatives L's that are functionally equivalent to L is expensive and potentially wasteful. Algorithm 3 shows our approach explained using the example from Section 2.

The procedure REFORMULATESERVICE (Algorithm 3) takes as argument D_L, the dependency matrix of the goal service (e.g., `e-Buyer`, Figure 4(a)), and S, the set of states of the component services (e.g., `Search-N-Ship`, Figure

Algorithm 2 Identifying Control Flow Dependency

```
1: procedure CTRLFLOWDEP(curr, Op)
2:   if (curr is a start-state) then
3:     return
4:   end if
5:   for all parent --g,a--> curr do
6:     if (OUTGOINGTRANS(parent) > 1) then
7:       C_{Op,g} := X
8:     end if
9:     CTRLFLOWDEP(parent, Op)
10:  end for
11: end procedure

12: procedure UPDATECTRLFLOWDEP(branchPoint)
13:   T := {paths from branchBegin to branchEnd}
14:   GuardSet := {g | branchBegin --g,a--> next}
15:   for all Op such that ∃g ∈ GuardSet : C_{Op,g} = X do
16:     if (∀t ∈ T : Op ∈ t) then
17:       ∀g' ∈ GuardSet : remove X from C_{Op,g'}
18:     end if
19:   end for
20:   for all g_1, g_2 ∈ GuardSet do
21:     C_{g_1,g_2} := Y
22:   end for
23: end procedure
```

Algorithm 3 Reformulation-based Service Composition

```
1: procedure REFORMULATESERVICE(D_L, S)
2:   if (D_L is null) then
3:     return true;
4:   end if
5:   R := {i | ∀j ∈ D_L : C_{i,j} is empty or only contains Y}
6:   select any r ∈ R do
7:     if (for any j ∈ D_L, C_{r,j} = Y) then
8:       DSet := REDUCE(D_L, r, true);
9:       for all D'_L ∈ DSet do
10:        if ¬REFORMULATESERVICE(D'_L, S) then
11:          break and backtrack to line 6
12:        end if
13:      end for
14:      return true
15:    else
16:      {D'_L} := REDUCE(D_L, r, false);
17:      if CREATETRANSITION(r, S) then
18:        if ¬REFORMULATESERVICE(D'_L, S') then
19:          backtrack to line 6
20:        else return true
21:        end if
22:      else backtrack to line 6
23:      end if
24:    end if
25:  end select return false
26: end procedure

27: procedure REDUCE(D, r, flag)
28:   DSet := ∅;
29:   if (flag = true) then
30:     for all (i ∈ {Op | C_{Op,r} = Y} ∪ {r}) do
31:       WorkingSet := {j | C_{i,j} = Y}
32:       DNew := D
33:       remove Y from all the C_{i,r} and C_{r,i} in DNew
34:       while (WorkingSet ≠ ∅) do
35:         for all j ∈ WorkingSet do
36:           if ∃k such that (C_{k,j} ⊈ ⋃{C_{k,l} | l ≠ j}) then
37:             add k to WorkingSet
38:           end if
39:           remove j from WorkingSet
40:           remove j-th row and column from DNew
41:         end for
42:       end while
43:       DSet := DSet ∪ DNew
44:     end for
45:   else
46:     remove r-th row and column from D
47:     DSet := {D}
48:   end if
49:   return DSet
50: end procedure
```

1(c) and Purchaser, Figure 1(d)). Initially, the procedure determines the set of operations in D_L which are not dependent on any other operation (line 5). These operations are required to be realized by the component services. For example, the operation a in the D_L (Figure 4(a)) of e−Buyer is not dependent on any other operation and can be realized by the transition from state t_0 to t_1 of Search−N−Ship (CREATETRANSITION[2] in line 17 holds true). As a result, D_L is updated to D_L^1 (Figure 4(b)) by removing the row and column corresponding to a signifying that a is already realized (removing the row) and all the dependencies on a are, therefore, eliminated (removing the column). This is achieved by executing lines 16 in REFORMULATESERVICE and 46−−47 in REDUCE (Algorithm 3). Next, REFORMULATESERVICE is recursively invoked (line 18) with D_L^1 and the new state-set S' of the component services reached after realizing operation a (e.g., Search−N−Ship is in state t_1). In D_L^1, operation b is not dependent on other operations and can be again realized by Search−N−Ship (line 17). Thus, D_L^1 is updated to generate D_L^2 (Figure 4(c)) following the same steps as described above and REFORMULATESERVICE is recursively invoked.

Proceeding further, in D_L^2 the only possible operations that can be considered are G_1, G_2 and G_3 since they are independent of other operations (line 5). However, all the three operations are guards and lead to different branches of a branch-point. Consequently, the component services must realize each individual branch. Furthermore, since the branches are mutually exclusive (i.e., their rows and columns are marked Y), if a branch G_i is considered, then all the branches at the same branch point corresponding to the guards G_j ($j \neq i$) must be removed from the dependency matrix. This is achieved when REFORMULATESERVICE at line 8 invokes REDUCE. The procedure REDUCE executes the statements from lines 29−−45 to create a set of matrices corresponding to each guard. During the execution of REDUCE, initially a working set of operations that must be removed is created (line 31). Referring to our example, consider the case where we are exploring the branch corresponding to guard G_1 (line 6) in D_L^2 , and the working set is $\{G_2, G_3\}$. Firstly, all the Y-marks are removed from the cells C_{G_i,G_j} (line 33) in D_L^2. Then for each operation x in the working set, any operation y that is *solely* dependent on x is added to the working set (line

[2]The procedure CREATETRANSITION is used to generate the alternate LTS specification, L', as part of the reformulation-based composition process. It takes as argument the operation r being analyzed and the set of component states S' reached after realization of r (by one of the component services) and generates a corresponding transition in L'. Details are present in [9].

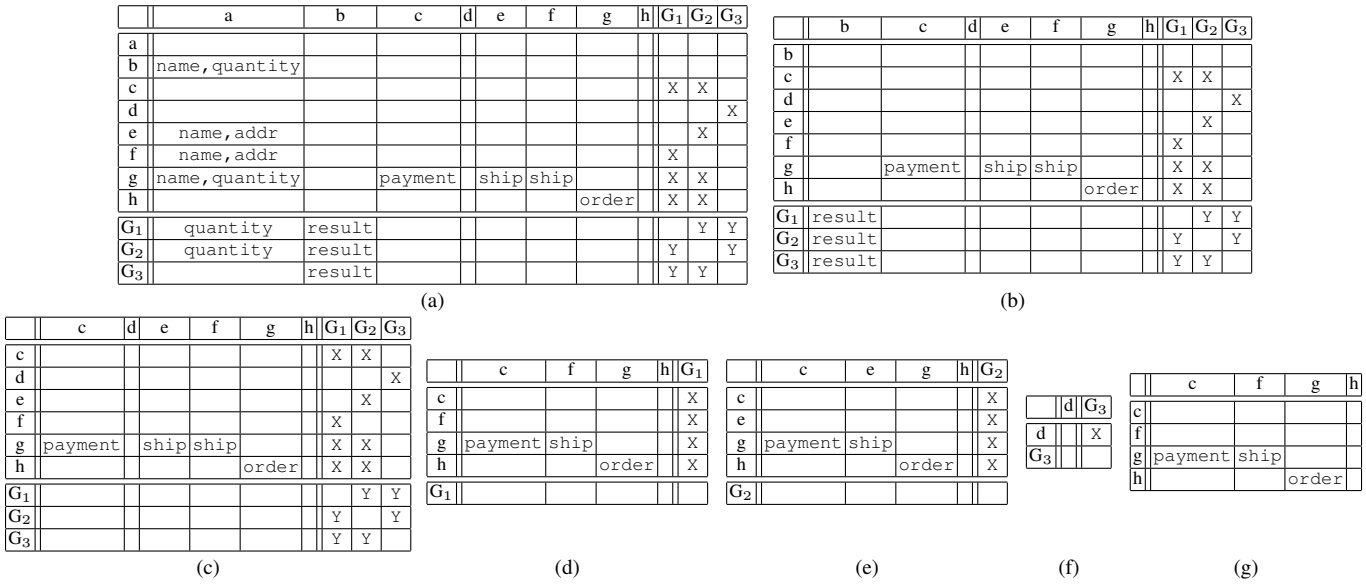


Figure 4: Dependency Matrices (a) D_L (b) D_L^1 (c) D_L^2 (d) D_L^3 (e) D_L^4 (f) D_L^5 (g) D_L^6

37). For example, for G_2 in D_L^2, the operation e is solely dependent on G_2, whereas operation c is not since $C_{c,G_1} = X$. Thus, e is added to the working set and operations solely dependent on e are identified iteratively for addition to the working set. On the other hand, since operation c is not solely dependent on G_2, it is not added to the working set. Furthermore at each iteration, an element is removed from the working set and its corresponding rows and columns are removed from the dependency matrix (line 40) and the above process continues until the working set becomes empty. In our example, execution of line 8 with D_L^2 will result in the creation of matrices D_L^3 (corresponding to G_1 being selected at line 30), D_L^4 (corresponding to G_2) and D_L^5 (corresponding to G_3) as shown in Figures 4(d), 4(e) and 4(f), respectively.

The procedure REFORMULATESERVICE will be invoked with each of these matrices as inputs (line 9). The procedure terminates successfully when the dependency matrix is empty denoting all operations are successfully realized by the component services and there were no failures during composition (lines 2--3). Otherwise, if a particular operation is not realizable then backtracking is performed to pick an alternate operation (line 11, 19, 22). For example, assuming that D_L^3 is selected in line 9, after realizing the guard G_1 ([quantity $<$ 100 $\wedge$ result = y]) by Search-N-Ship, it will be updated to create a new dependency matrix D_L^6 (Figure 4(g)). In D_L^6, the operations that can be considered are c (?inPay(payment)) and f (NormalShip (name,addr;ship)) since they are independent of other operations (line 5). However, if operation c is selected first in line 6, it will result into a composition failure since such a behavior cannot be realized by any of the existing component services. That is, none of the component services (Figures 1(c) & 1(d)) has a transition associated with the guard G_1 immediately followed by another transition associated with the action c. As a result, REFORMULATESERVICE will backtrack, and select f and determine if it can be realized. Thus, in essence, where the existing algorithms for service composition would have failed at this point, our approach automatically adapts the goal service based on the analysis of control and data flow dependencies for identifying feasible compositions. For this particular example, the composition obtained eventually will correspond to the (alternate) goal service e-Buyer' (Figure 3). Note that even though the original goal service e-Buyer and its alternate model e-Buyer' are structurally different, they are functionally equivalent with respect to data and control dependencies.

Theorem 2 (Soundness & Completeness)**.** *Given a target L and set of component services CS with start state-set S, there exists a service L' such that* REFORMULATESERVICE *(D_L, S) returns true iif $L \equiv L'$ and CS realizes L'.* □

Complexity Analysis. The algorithms DATAFLOWDEP and CTRL FLOWDEP perform backward depth-first traversal from each state in an LTS and their complexity is $O(|S| \times | \longrightarrow |)$ where $|S|$ and $| \longrightarrow |$ are the number of LTS states and transitions respectively. The procedure UPDATECTRLFLOWDEP considers all possible paths of branches. The algorithm can be written by memorizing (recording in CTRLFLOWDEP) the set of operations that are possible from every state. In that case, the complexity becomes same as that of backward depth-first traversal.

Algorithm 3 can be also made efficient by memorizing the arguments used for invoking REFORMULATESERVICE such that calls with the same arguments are not made re-

peatedly. The exploration of the state-space is done in a depth-first fashion (complexity linear to the number of transitions). At each depth, the complexity is determined by the procedure CREATETRANSITION ([9]) used for realizing an operation from the component services.

5 Related Work

There is a recent body of work that has focused on the notion of "adaption" (or change) in Web services in a broader sense. Nezhad et al. [7] describe the use of a mismatch tree to capture mismatches of various types (e.g., signature, messages) between the specification and available component services, to solicit information needed to resolve the mismatches from the service developers, and to generate the necessary adapters. Brogi and Popescu [3] have proposed a similar approach for generation of BPEL adapters. On a different note, Harney and Doshi [5] have proposed a technique that monitor changes in dynamic properties of a component service (e.g., performance) to determine whether to replace it with another service that offers the same functionality within a composite service. Similarly, Chafle et al. [4] have proposed a framework that specifies several alternate plans to choose from in response to changes in the environment (e.g., performance, cost, availability of component services).

Unlike the work summarized above, the emphasis of this paper is on neither resolving mismatches between components by generating adapters nor adapting composite services in response to changes in the environment. Instead, our focus is on generate a composite service from the available component services in settings where the composition algorithm fails because the goal service LTS, as specified, cannot be realized using available components, but a functionally equivalent reformulation can be realized. Thus, in essence, our work on "adaptation/change" is complementary to the existing work.

6 Summary and Discussion

We propose a framework for composing Web services through automatic reformulation of service specifications (represented as labeled transition systems) based on the analysis of control and data flow dependencies. We introduce a data structure called dependency matrix to support this analysis and show that any goal LTS reformulation that does not violate the data and control dependencies that are implicit in the specified goal service LTS specification is *provably functionally equivalent* to the specified goal service. We have developed an efficient algorithm that is linear in the size of the goal LTS specification, and can generate an (as yet untried) alternatives that are *functionally equivalent* to the user-supplied goal LTS. This process proceeds until a composite service is obtained (i.e., composition succeeds) or no alternative reformulations remain to be tried. The resulting framework can help limit in the need for the tedious manual intervention in reformulating specifications by limiting such interventions to settings where neither the specified goal LTS nor any of its its functionally equivalent reformulations (that conform to the data and control flow dependencies implicit in the goal specification) can be realized using the available component services.

Some interesting directions along this line of research includes consideration of more expressive specifications than those captured by LTSs. Our current framework represents services using LTSs, which are essentially discrete-event systems. Some application scenarios require modeling of actions that extend over temporal intervals (e.g, duration of action `a` spans duration of action `b`). In this context, interval-based temporal representations (e.g., temporal algebra) would be interesting to explore. We are also currently working towards developing benchmarks and performing empirical evaluation aimed at assessing the extent to which the our framework eliminates the need for manual intervention in real-world service composition scenarios.

References

[1] B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[2] D. Berardi, D. Calvanese, D. G. Giuseppe, M. Lenzerini, and M. Mecella. Automatic Service Composition based on Behavioral Descriptions. *International Journal on Cooperative Information Systems*, 14(4):333–376, 2005.

[3] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *4th International Conference on Service-Oriented Computing*, pages 27–39. LNCS 4294, 2006.

[4] G. Chafle, P. Doshi, J. Harney, S. Mittal, and B. Srivastava. Improved Adaptation of Web Service Compositions using Value of Changed Information. In *5th IEEE International Conference on Web Services*, pages 784–791. IEEE CS Press, 2007.

[5] J. Harney and P. Doshi. Speeding Up Adaptation of Web Service Compositions Using Expiration Times. In *16th World Wide Web Conference*, pages 1023–1032. ACM Press, 2007.

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

[7] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *16th World Wide Web Conference*, pages 993–1002. ACM Press, 2007.

[8] J. Pathak, S. Basu, and V. Honavar. Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements. In *4th International Conference on Service Oriented Computing*, pages 314–326. LNCS 4294, Springer-Verlag, 2006.

[9] J. Pathak, S. Basu, R. Lutz, and V. 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.

[10] J. Rao, P. Kungas, and M. Matskin. Logic-based Web Services Composition: From Service Description to Process Model. In *2nd IEEE International Conference on Web Services*, pages 446–453. IEEE CS Paper, 2004.

Appendix

Theorem 1. *For any two LTSs L and L', $L \equiv L'$ if and only if their dependency matrices D_L and $D_{L'}$ are identical.*

Proof: Let $L \equiv L'$ and $D_L \neq D_{L'}$. D_L and D'_L must have the same set of row or column labels since from Definition 3, $L \equiv L'$ implies that they have same set of operations. Therefore, $D_L \neq D'_L$ implies that there exists $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ where $\mathtt{C}_{i,j} \in D_L$ and $\mathtt{C}'_{i,j} \in D'_L$. Note that, $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ implies that $\mathtt{C}_{i,j} \cap \mathtt{C}'_{i,j} = \emptyset$. Therefore, the only way $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ is when $\mathtt{C}_{i,j} \neq \emptyset$ and $\mathtt{C}'_{i,j} = \emptyset$ or vice versa. Let $\mathtt{C}_{i,j} = \{\mathtt{X}\}$ and $\mathtt{C}'_{i,j} = \emptyset$ (case 1 in Definition 4). This implies that i is control dependent on j in L and i is not control dependent on j in L'. Therefore, in all sequences in $\mathcal{B}(L)$ which contains i, j appears before i while there exists some sequence in $\mathcal{B}(L')$, where i is present and j is absent or i is present before j. This contradicts our initial assumption that $L \equiv L'$ by Definition 3. Same type of contradiction can be realized for other cases where $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$.

Next assume $D_L = D'_L$ and $L \not\equiv L'$. Therefore, there exists a pair of operations i and j such that j appears before i in all sequences in $\mathcal{B}(L)$ containing i. However, there exists at least one sequence in $\mathcal{B}(L')$, containing i and j, where i appears before j. The case implies that i depends on j in L while it is not dependent on j in L'. In other words, $\mathtt{C}_{i,j} \neq \emptyset$ while $\mathtt{C}'_{i,j} = \emptyset$. This leads to contradiction of the initial assumption of $D_L = D'_L$. □

Theorem 2. *Given a service L and set of component services CS with start state-set S, there exists a service L' such that* REFORMULATESERVICE *(D_L, S) returns true if and only if $L \equiv L'$ and CS realizes L'.*

Proof: Let REFORMULATESERVICE(D_L, S) return true and for all L's realized from CS, such that $L \not\equiv L'$. From Theorem 1, $\forall L' : L' \not\equiv L \Rightarrow D'_L \neq D_L$. In other words, there exists at least one pair of operations in L such that $\mathtt{C}_{i,j} \neq \mathtt{C}'_{i,j}$ ($\mathtt{C}_{i,j} \in D_L$ and $\mathtt{C}'_{i,j} \in D'_L$) that is not realizable from CS. Proceeding further, $\mathtt{C}_{i,j}$ demands a specific ordering or mutual exclusion of i and j in all sequences and this is not realizable from CS. This, in turn, implies that REFORMULATESERVICE fails at `line 17` for all possible choices at `line 6`, and eventually returns false at `line 25`. This leads to contradiction of our initial assumption that REFORMULATESERVICE returns true.

Next, consider that case where REFORMULATESERVICE returns true but CS *does not* realize any L' ($\equiv L$). I.e., for all possible alternate sequences in $\mathcal{B}(L)$, their exists some operation in each sequence for which CREATETRANSITION fails. If such a failure occurs (`line 17`) in REFORMULATESERVICE, the algorithm backtracks and selects alternate functionally equivalent sequences using D_L. Finally, when all alternates are exhausted and CREATETRANSITION fails in all of them, REFORMULATESERVICE returns false (`line 25`). This contradicts our initial assumption that REFORMULATESERVICE returns true.

Finally, consider that there exists an $L \equiv L'$ and CS realizes L' but REFORMULATESERVICE(D_L, S) returns false (`line 25`). This will happen when CREATETRANSITION fails for all possible alternates identified by REFORMULATESERVICE. If we assume that REFORMULATESERVICE correctly computes all possible alternates, then failure of REFORMULATESERVICE due to failure of CREATETRANSITION directly contradicts the initial assumption that CS realizes $L'(\equiv L)$.[3]

The other alternate is that REFORMULATESERVICE does not correctly consider all possible alternates, and hence fails to identify the $\mathcal{B}(L')$ which is realizable from CS. `Line 6` in REFORMULATESERVICE considers all operations which are not data-dependent on any other operation as candidates for realizability. If such a candidate operation is a guard, REFORMULATESERVICE invokes REDUCE to obtain dependency matrices of all paths beyond the branch point of the guard under consideration. The procedure REDUCE selects all the mutually exclusive guards (`line 30`) and for a particular guard i, it firstly removes rows and columns of the guards that cannot appear in the same sequence as i, and then iteratively removes the rows and columns of operations that are solely dependent (directly or indirectly) on these guards (`lines 31--42`). In short, REDUCE correctly identifies all the possible dependency matrices beyond a branch point and REFORMULATESERVICE, in turn, considers all the possible ways of realizability using those matrices. Therefore, if there exists an $L' \equiv L$ which is realized from CS, then the REFORMULATESERVICE must return true. □

[3] We are assuming the correctness of CREATETRANSITION. Details of its correctness can be obtained in [8, 9].