

Local and On-the-fly Choreography-based Web Service Composition *

Saayan Mitra¹ Samik Basu² Ratnesh Kumar¹

¹Department of Electrical and Computer Engineering, ²Department of Computer Science
Iowa State University, Ames, IA 50011-1040, USA
{saayan, sbasu, rkumar}@iastate.edu

Abstract

We present a goal-directed, local and on-the-fly algorithm for verifying the existence of and synthesizing a choreographer for Web service composition. We use i/o-automata to represent the (existing/component) services, the desired functionality of the composition, and a choreographer to achieve the desired service by composing the existing ones [7]. As is the case with our prior work [7], choreographer existence and synthesis are typically performed by identifying all possible compositions realizable from the existing services and verifying whether one such composition conforms to the desired required functionality. Such a technique is subject to the state-space explosion as the computation of compositions is exponential to the number of components. In light of this, we have developed a tabled-logic programming technique which generates and explores compositions of the component services in a goal-directed fashion to prove or disprove the existence of choreographer and to infer whether the desired functionality is realizable. We present a prototype implementation and show the practical applicability of our goal-directed, local and on-the-fly technique using a variety of composition problems along with the corresponding computational savings in terms of the number of the states and transitions explored.

1 Introduction

Web service composition problem involves identifying whether a set of existing services can be employed in a collaborative fashion to realize a desired goal service. A typical service represented using service standards like WS-BPEL, OWL-S, WSDL [8] describes the input/output behavior of the service, i.e., given a set of inputs a service computes and produces the corresponding outputs. To realize a goal service from an existing set of services, one has to identify a specific way the existing services can be connected (output from one provided as input to others) such that the desired input/output behavior of the goal service can be realized. The desired connections between existing services are provided by a mediator referred to as chore-

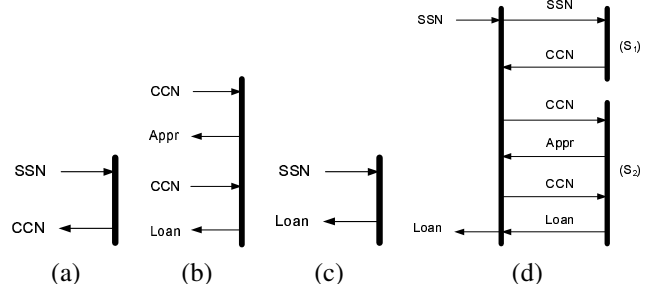


Figure 1: (a) $Serv_1$, (b) $Serv_2$, (c) Goal $Serv_G$ and (d) Composed Services.

ographer. The aim of service composition is to synthesize such a choreographer, when possible.

Driving Problem. As is the case with our prior work [7], the problem of existence and synthesis of choreographer is typically solved by first generating all possible composite behaviors of existing services and identifying a subset to be achieved by a choreographer to realize the desired functionality of the goal service. In [7] we showed that the problem reduces to generating an i/o-automaton describing all possible composite behaviors, called the *universal-service automaton*, and verifying whether it *simulates* the i/o-automaton representation of the goal service. Computation of the universal service-automaton leads to the state-space explosion since the computation of all possible behaviors of composition is exponential to the number of the participants in the composition, which limits its scope for practical applications. Since only those behaviors of the composition that simulate the goal service are of interest, it makes sense to restrict the generation and exploration of the universal-service automaton that can realize the goal. This is the motivation behind the goal-directed generation and exploration (of the universal-service automaton) proposed in this paper. Only the part of the universal-service automaton relevant for achieving the goal service is generated (making the approach on-the-fly) and explored (making the approach local).

Consider input/output sequence-diagrams of services in Figure 1(a,b) where $Serv_1$ takes as input SSN (Social Security Number) and produces a corresponding CCN (Credit Card Number) while $Serv_2$ uses CCN to provide Appr (credit approval) and Loan (loan approval). The desired functionality $Serv_G$ is presented in Figure 1(c) which takes as input

*The research was supported in part by the National Science Foundation under the grants NSF-ECS-0218207, NSF-ECS-0244732, NSF-EPNES-0323379, NSF-0424048. and NSF-ECS-0601570.

SSN and outputs Loan. A choreographer that can communicate with Serv_1 and Serv_2 to realize Serv_G is presented in Figure 1(d); the left vertical column describes the input/output sequence behavior of the choreographer. Observe that, the choreographer first communicates with Serv_1 and then with Serv_2 . There are many other combinations in which Serv_1 and Serv_2 can communicate. For example, the choreographer can communicate in reverse order—first with Serv_2 and next with Serv_1 . However, such combinations are not of interest from the perspective of the goal Serv_G and exploring such possibilities results in unnecessary overhead in computation time and space.

Our Solution. We develop a goal-directed technique which only generates and explores parts of possible composite behaviors that are required to prove or disprove the realizability of the desired goal service (using simulation relation) and the existence of a corresponding choreographer. At its core, our technique relies on logical encoding of the composition problem in tabled-logic programming environment. We show that such encoding is natural and straightforward. Furthermore, given the goal service specification, evaluation of logic program allows exploring communications between choreographer and component services that are relevant for achieving the goal service and results in a local and on-the-fly choreographer synthesis technique.

Contributions.

1. We present a logical encoding of service composition and choreographer existence problem which is natural and straight-forward in the sense that it directly encodes the definitions of composition and choreography as logical relations.
2. We show that goal-directed evaluation of our logical encoding results in local and on-the-fly computation of service composition.
3. We present preliminary experimental evaluation using a variety of composition problems which provides strong testimony of the effectiveness of our technique.

Organization. Section 2 presents the overview of the composition problem in the setting of input/output automata and preliminaries on logic programming. Section 3 describes the logical encoding of the composition problem along with the actual implementation followed by Section 4 which presents the experimental results. Section 5 discusses the related work. Finally, we conclude with some future directions of research in Section 6.

2 Background

2.1 I/O-automata based Composition

We describe the formulation of the composition problem in terms of i/o-automata and simulation relations, and provide an overview of our solution approach from [7], which

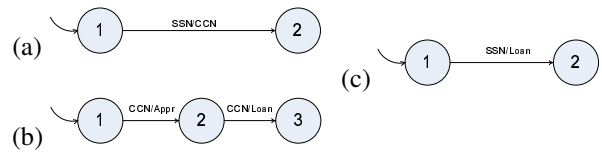


Figure 2: Automata: Component Services (a) A_1 , (b) A_2 . Goal Service (c) A_G .

forms the basis of our proposed logical encoding that performs a local and on-the-fly computation of the composition.

Definition 1 (I/O Automaton) An i/o automaton A is defined by a tuple (S, S^0, I, O, Δ) , where, S is the set of states, $S^0 \subseteq S$ is the set of initial states, I is the set of inputs, O is the set of outputs and $\Delta \subseteq S \times I \times O \times S$ is the set of transitions. An element of Δ , represented by $\delta = (s, i, o, s')$, is such that $s \in S$ is the origin state of the transition, $i \in I$ is the input to the transition, $o \in O$ is the output of the transition, and $s' \in S$ is the destination state of the transition. We will often use $s \xrightarrow{i/o} s'$ to denote $(s, i, o, s') \in \Delta$.

Figure 2(a,b,c) presents the i/o-automata models for the component and goal services and described in Figure 1(a,b,c). The start states have arrows pointed to them.

The main functionality of a choreographer is to relay the client inputs to the appropriate component services and relay the component outputs either to other components or the client. The choreographer also stores past inputs/outputs it has seen from the client or the component services to *induce* future inputs to components or outputs to client. A *history* information is maintained to keep track of past inputs/outputs seen by a choreographer. The following definition captures this history for all possible interleaved behaviors of the components.

Definition 2 (Interleaving Product with History)

Given a set of service automata A_1, A_2, \dots, A_N where $A_n = (S_n, S_n^0, I_n, O_n, \Delta_n)$ for $1 \leq n \leq N$, the interleaving product with history is defined to be the automaton $\parallel_n^H A_n = (S_H, S_H^0, I, O, \Delta_H)$ where $S_H \subseteq S_1 \times S_2 \times \dots \times S_N \times 2^{I \cup O}$ where $I = \bigcup_{n \leq N} I_n$

and $O = \bigcup_{n \leq N} O_n$; $S_H^0 = S_1^0 \times S_2^0 \times \dots \times S_N^0 \times \{\emptyset\}$ and

$\Delta_H \subseteq S_H \times I \times O \times S_H$ such that

$$(s_1, \dots, s_N, h) \xrightarrow{i/o} (s'_1, \dots, s'_N, h') \in \Delta_H \Leftrightarrow \left[\begin{array}{l} \exists n \leq N : s_n \xrightarrow{i/o} s'_n \in \Delta_n \wedge \\ \forall m \leq N, m \neq n : s'_m = s_m \wedge h' = h \cup \{i, o\} \end{array} \right]$$

Figure 3(a) presents the interleaving product with history of component services in Figure 2(a,b). Each state is annotated with label (s_n, t_m, h) where s_n denotes the state of

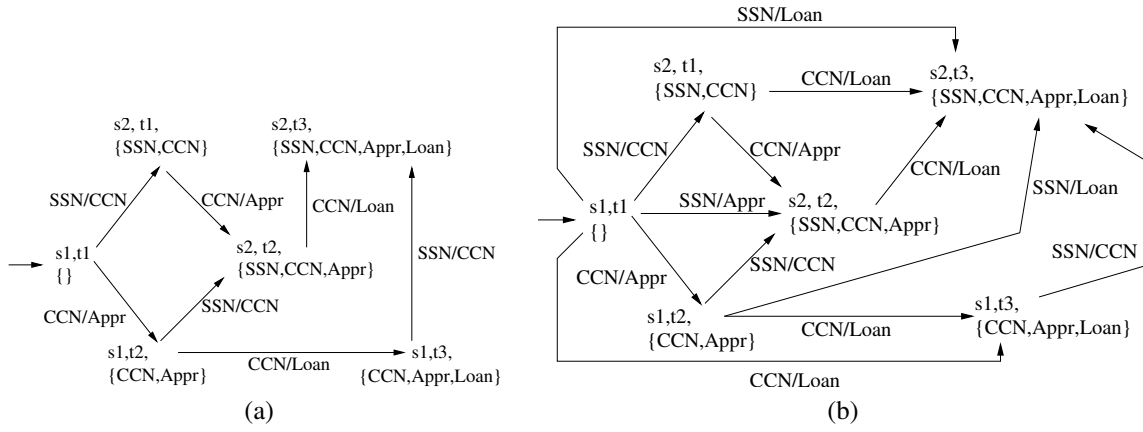


Figure 3: (a) $\|_n^H A_n$ -automaton and (b) \cup^T -automaton.

A_1, t_m denotes the state of A_2 and h is the set of past inputs/outputs available for inducement when A_1 and A_2 are at state s_n and t_m respectively. Observe that when A_1 and A_2 are at states s_1 and t_1 (start states), the history set is empty. When A_1 and A_2 are at states s_2 and t_1 , the choreographer can induce SSN and CCN as SSN is an input provided by the client while CCN is the output generated by A_1 .

Given, the history information at each state, the choreographer can use these information in subsequent steps to provide inputs/outputs (from the history set) to the component services without depending on the client or components. The behavior that can be achieved by the choreographed components is defined using *transduced-closure* as follows:

Definition 3 (Transduced-Closure of Automaton) Given an automaton with history $A_H = (S_H, S_H^0, I, O, \Delta_H)$, the transduced-closure of A_H is the automaton $A_H^T = (S_H, S_H^0, I, O, \Delta_H^T)$ where $(s, h) \xrightarrow{i/o} (s', h') \in \Delta_H^T$ if and only if

$$\exists k : \left[\begin{array}{l} (s, h) \xrightarrow{i/o_1} (s_1, h_1) \in \Delta_H \wedge \\ (s_1, h_1) \xrightarrow{i_1/o_2} (s_2, h_2) \in \Delta_H \wedge \dots \\ (s_{k-1}, h_{k-1}) \xrightarrow{i_{k-1}/o_k} (s_k, h_k) \in \Delta_H \wedge \\ (\forall 1 \leq j < k : i_j \in h_j) \wedge (o_k = o) \end{array} \right] \quad (1)$$

The universal-service automaton, denoted \cup^T , is defined to be the transduced-closure of $\|_n^H A_n$, i.e., $\cup^T := (\|_n^H A_n)^T$, where $\{A_n\}$ is the set of component services.

Figure 3(b) presents the transduced closure of Figure 3(a). Notice that the choreographer can utilize the history at state (s_2, t_1) to produce CCN input for A_2 which outputs $Appr$. Therefore, if the client provides SSN , the choreographed components can produce $Appr$. This is represented by the transition from (s_1, t_1) to (s_2, t_2) in Figure 3(b).

\cup^T -automaton contains all possible choreographed behavior of the services. Therefore, the goal automaton A_G is realizable from the services using some choreographer if and only if all possible behavior of A_G is also present

in \cup^T -automaton. This is verified using simulation relation defined as follows:

Definition 4 (Simulation [6]) Given an i/o automaton $A = (S, S^0, I, O, \Delta)$, for all s_1 and s_2 in S , s_1 is simulated by s_2 if and only if they are related by the largest simulation relation denoted by $s_1 \sqsubseteq s_2$ and defined as: $s_1 \sqsubseteq s_2 \Rightarrow [\forall t_1 : s_1 \xrightarrow{i/o} t_1 \Rightarrow (\exists t_2 : s_2 \xrightarrow{i/o} t_2 \wedge t_1 \sqsubseteq t_2)]$.

An i/o automaton $A_1 = (S_1, S_1^0, I_1, O_1, \Delta_1)$ is said to be simulated by $A_2 = (S_2, S_2^0, I_2, O_2, \Delta_2)$, denoted by $A_1 \sqsubseteq A_2$, iff all states in S_1^0 are simulated by some state in S_2^0 .

We established the following theorem in [7] which states that a transduced-choreographer exists if and only if the goal service is simulated by the universal-service automaton.

Theorem 1 Given a goal A_G and a set of services A_1, A_2, \dots, A_N , the goal can be realizable from the composition of A_n s with a transducing choreographer if and only if $A_G \sqsubseteq \cup^T$ where \cup^T is the transduced-closure of the $(\|_n^H A_n)$ -automaton obtained by taking interleaving product with history of the automata $\{A_n\}$ s.

2.2 Preliminaries on Logic Programming

XSB [16], developed at SUNY, Stony Brook is based on Prolog-style SLD resolution with tabling. Tabling, which is essentially a memorizing technique, allows XSB to compute least model solutions of normal logic programs and avoid repeated subcomputations. XSB relations/predicates are defined as

$R :- R_1, R_2, \dots, R_n.$
 $S :- S_1; S_2; \dots; S_n.$

where relation R evaluates to true if *all* (conjunction) the relations R_1, R_2, \dots, R_n evaluates to true. On the other hand, relation S evaluates to true if *at least one* of (disjunction) the relation S_1, S_2, \dots, S_n evaluates to true. A relation with no rhs of $:-$ is referred to as fact.

Consider a simple encoding of a graph in XSB. The edges in the graph are defined as *edge-facts* of the form as

shown in Figure 4(a). The reachability between states in the above graph can be encoded in XSB as `reach`-relations (Figure 4(a)). The predicate `reach` is defined using two rules. The first rule states that a state is reachable from another if there is an edge between the two. Observe that, `S` and `T` are variables¹ which are existentially quantified. The second rule computes the transitive closure: a state `T` is reachable from `S` if there is an edge from `S` to `S1` and `T` is reachable from `S1`.

If we want to find all the reachable states from `s0`, we evaluate the query `reach(s0, Ans)` using the above program; on termination, the variable `Ans` evaluates (bounded) to state reachable from `s0`. Figure 4(b) presents the execution tree of the above query. Using the first rule, `reach(s0,s0)` and `reach(s0, s1)` evaluates to true. However, evaluation of the second rule fails to terminate because `reach(s0, Ans)` invokes `edge(s0,s0)` followed by `reach(s0, Ans)`. This can be avoided by using tabling feature of XSB and using the directive `:- table reach/2`. The directive ensures that the least model of the relation `reach` is computed. In the current context, `reach(s0,Ans)` will fail if it leads to invocation of `reach(s0, Ans)`. Whenever a failure occurs along an execution path, XSB backtracks automatically to the last choice point to evaluate and execute along an alternate path if possible. In Figure 4(b), `reach(s0, Ans)` leads to `edge(s0, S1)`, `reach(S1, Ans)` which grounds `S1` to `s0` as `edge(s0,s0)` is a fact and fails on `reach(s0, Ans)`. It then backtracks and selects another fact from edge: `edge(s0, s1)` and continues with the evaluation of `reach(s1, Ans)`. If there is no choice point available, the predicate execution terminates and returns the evaluation of variable (in this case `Ans`) if there is one; otherwise the predicate is said to be unsatisfiable. The invocation `reach(s0, Ans)` terminates and correctly produces the results for `Ans`.

Observe that, states that are not reachable from `s0` are never explored in the above execution. This is because the execution of any recursive `reach` predicate occurs in a goal-directed fashion for a specific valuation of the first argument (source state).

3 Local and On-the-fly Algorithm

3.1 Logical Encoding

Encoding I/O Automata. The i/o automaton for a service is described as logic program using facts describing the transitions and the start state. For example, automata A_1 and A_2 from Figure 2(a,b) are encoded as:

```
servicetrans(s1, (ssn, ccn), s2).
startservice(s1).    %% start state of A1

servicetrans(t1, (ccn, appr), t2).
servicetrans(t2, (ccn, loan), t3).
startservice(t1).    %% start state of A2

startcompose([s1,t1]). %% for interleaving product
```

¹Upper-case alphabets denote logical variables and lower case alphabets denote constants.

where `s1` and `s2` are the states in A_1 and `t1`, `t2` and `t3` are states in A_2 . Similarly, the goal i/o automaton (Figure 2(c)) is encoded as:

```
goaltrans(g1, (ssn, loan), g2).
startgoal(g1).
```

Encoding Interleaving Product. The interleaving product of the services (Definition 2) is encoded as XSB relations:

```
partrans(([St|Sts], Hist), (I, O),
         ([NewSt|Sts], NewHist)) :-
    servicetrans(St, (I, O), NewSt),
    append([I, O], Hist, NewHist).

partrans( ([St|Sts], Hist), (I, O),
         ( [St|NewSts], NewHist) ) :-
    partrans( (Sts, Hist), (I, O), (NewSts, NewHist)).
```

There are two rules defining the relation `partrans`. First, consider the Rule 1 and the first argument of `partrans`. The first argument denotes tuple: list of service states and the associated history. List in XSB is represented as `[St|Sts]` where `St` represents the first element (head) of the list and `Sts` represents the rest (tail) of the list. Therefore Rule 1 states that there exists a transition in the interleaving product if the service-state present at the head of the list has a transition and the rest of service-states remain unaltered. Furthermore, the history set `Hist` is updated to `NewHist` to include `I` and `O` using the `append` predicate. Rule 2, on the other hand, states that there exists a transition in the interleaving product if any service-state in the tail of the list has a transition. Note that, we just encoded the rules for transition relation of the interleaving product and do not actually generate the product graph.

Encoding Transduced Closure. The transduced closure transition relation (Definition 3) is similarly encoded as follows:

```
closuretrans((Sts, Hist), (I, O),
             (NewSts, NewHist)) :-
    partrans((Sts, Hist), (I, O), (NewSts, NewHist)).

closuretrans((Sts, Hist), (I, O),
             (NewSts, NewHist)) :-
    partrans((Sts, Hist), (I, _), (StsTemp, HistTemp)),
    histtrans((StsTemp, HistTemp), (_, O),
             (NewSts, NewHist)).
```

The first rule states that there is a transduced closure transition if there is an interleaved product transition. The second rule states that a transduced closure transition exists if there is an interleaved product transition followed by a sequence of transitions (computed by `histtrans` described below) where inputs are provided from the history set `HistTemp`. The notation “`_`” denotes variable whose valuation is not captured in the evaluation. Observe that, `I` in `closuretrans` is obtained from `partrans` and `O` is obtained from `histtrans`. The rule follows directly the Definition 3 where `histtrans` is computing $\exists k : (s_1, h_1) \xrightarrow{i_1/o_2}$

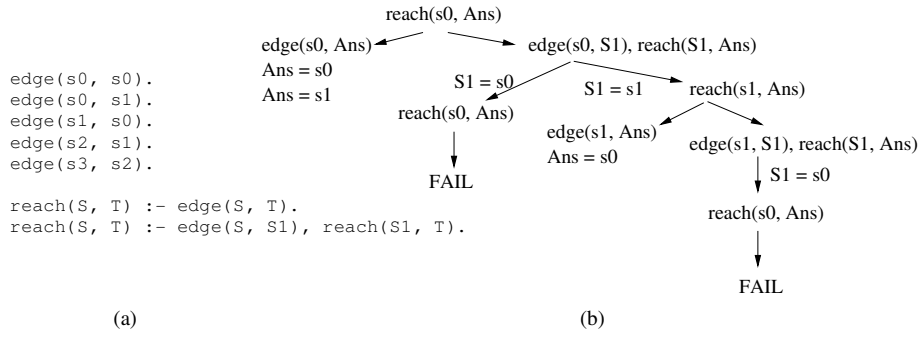


Figure 4: (a) Reachability in XSB. (b) Evaluation tree for $reach(s_0, Ans)$.

$(s_2, h_2) \xrightarrow{i_2, o_3} \dots \xrightarrow{i_{k-1}, o} (s_k, h_k) \wedge \forall 1 \leq j \leq k : i_j \in h_j$
(Equation 1). The definition is explained in details below:

```
histtrans((Sts, Hist), (I, O), (NewSts, NewHist)) :-
  partrans((Sts, Hist), (I, O), (NewSts, NewHist)),
  member(I, Hist).
```

```
histtrans((Sts, Hist), (I, O), (NewSts, NewHist)) :-
  partrans((Sts, Hist), (I, _), (StsTemp, HistTemp)),
  member(I, Hist),
  histtrans((StsTemp, HistTemp), (_, O),
            (NewSts, NewHist)).
```

The rules are similar to `closuretrans`. The first rule corresponds to the base case: $(s_{k-1}, h_{k-1}) \xrightarrow{i_{k-1}, o} (s_k, h_k)$ while the second rule corresponds to the transitivity. In each rule, the inputs of `partrans` are provided by the `Hist` (predicate `member(I, Hist)` evaluates to true if `I` is present in `Hist`).

Encoding Simulation Relation. The final step of the choreographer existence problem is to verify simulation of the goal automaton by the transduced closure automaton. Recall from Definition 4, two states are said to be simulation equivalent if and only if they are related by the largest simulation relation \sqsubseteq : $s_1 \sqsubseteq s_2 \Rightarrow [\forall t_1 : s_1 \xrightarrow{i/o} t_1 \Rightarrow (\exists t_2 : s_2 \xrightarrow{i/o} t_2 \wedge t_1 \sqsubseteq t_2)]$. However, evaluation of logic program results in the least model computation. In order to encode simulation as logic program, we consider the negation of the above relation, i.e., two states are said to be not simulation equivalent if and only if they are related by the least not-simulation relation $\not\sqsubseteq$ (dual of \sqsubseteq) defined as follows:

$$s_1 \not\sqsubseteq s_2 \Leftarrow [\exists t_1 : s_1 \xrightarrow{i/o} t_1 \wedge (\forall t_2 : s_2 \xrightarrow{i/o} t_2 \Rightarrow t_1 \not\sqsubseteq t_2)] \quad (2)$$

Therefore, from Theorem 1 a goal A_G is *not realizable* from choreographed U^T if and only if $A_G \not\sqsubseteq U^T$. The above equation can be directly encoded in XSB as:

```
:- table nsim/2.
nsim(S1, S2) :-
  goaltrans(S1, A, T1),
  forall(closuretrans(S2, A, T2), nsim(T1, T2)).

forall(closuretrans(S2, A, T2), nsim(T1, T2)) :-
  closuretrans(S2, A, T2),
```

```
(nsim(T1, T2) -> fail; !, fail).
forall(_, _).
```

The predicate `nsim(S1, S2)` corresponds to $s_1 \not\sqsubseteq s_2$ relation. It evaluates to true when there exists a transition in the goal A_G from s_1 which is not simulated by any transition (denoted by `closuretrans`) in U^T from s_2 .

The predicate `forall` corresponds to $\forall t_2 : s_2 \xrightarrow{i/o} t_2 \Rightarrow t_1 \not\sqsubseteq t_2$ in Equation 2. The evaluation of `forall` can be explained as follows. It identifies a matching `closuretrans` (on same i/o as the goal transition) and verifies whether `nsim(T1, T2)` holds true. If `nsim(T1, T2)` evaluates to true, it *fails* and backtracks to obtain another result for `closuretrans`; otherwise it fails and does not backtrack—backtracking is terminated using the operator “!”, referred to as *cut*. If `closuretrans` evaluates to false, then predicate `forall` succeeds via the second rule, in which case the instance of `nsim` that invoked the `forall` also succeeds.

Given the service and goal automata, the existence of choreographer is verified by invoking `nsim(GoalState, (ServiceStates, []))` where `startgoal(GoalState)` and `statecompose(ServiceStates)` evaluates to true. The empty list `[]` associated with `ServiceStates` denotes that the history-set is empty initially. If `nsim(GoalState, (ServiceStates, []))` evaluates to true then the choreographer does not exist; otherwise it does.

Once the existence of a choreographer is established, we analyze the transitions explored to prove the existence and the states in the U^T -automaton that are similar to states in goal automaton to synthesize the choreographer. This ensures that synthesis of choreographer also explores only the transitions that are required to prove its existence.

3.2 Example Run

Consider the example in Figure 2. In order to verify the existence of a choreographer, we invoke `nsim(g1, ([s1, t1], []))` where `g1` is the start state of the goal and `([s1, t1], [])` is the start state of U^T -automaton corresponding to the services. The execution trace is presented in Figure 5. Each step shows the expansion of a predicate by the rhs of its rule. Some expansions are shown with grounding of variable when a predicate is satisfied. For example `nsim`, at the root, is expanded to conjunction of predicates `goaltrans` and `forall`. The predicate

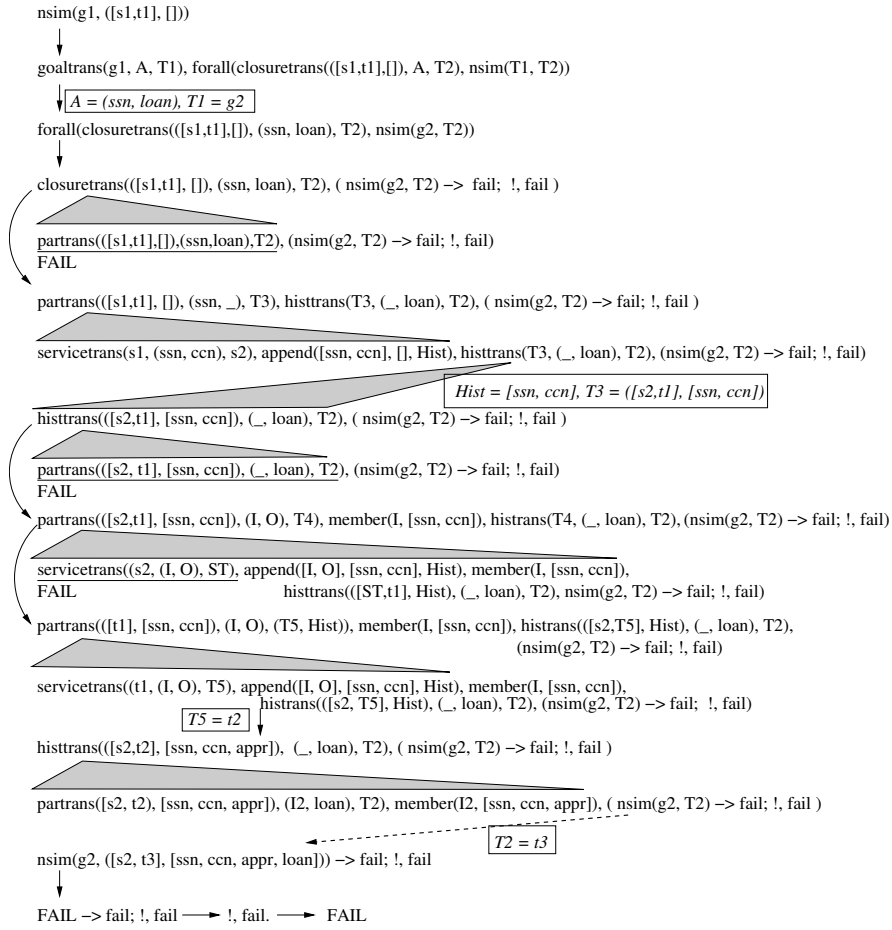


Figure 5: Execution tree.

`goaltrans` evaluates to true and grounds the valuation of the variable A to $(ssn, loan)$ and $T1$ to $g2$ (corresponding to the goal transition). Whenever a failure is encountered, backtracking is performed from the last choice point. For example, first invocation of `closuretrans` fails due to failure to satisfy `partrans`—there is no transition defined from $((s1, t1), [])$ on input ssn and output $loan$. As a result, prolog engine backtracks to obtain an alternate way to satisfy `closuretrans` via the second rule of `closuretrans`. Finally, a query to predicate `nsim(g2, ([s2, t2], [ssn, ccn, appr, loan]))` is made which fails as there is goal-transition from $g2$. This leads the execution to evaluate “!, fail”, i.e., go over a cut and as mentioned above backtracking is stopped. This implies that for a goal transition sequence, a sequence of transitions in the U^T -automaton is obtained that simulates the goal transition sequence. his leads to As a result, `nsim(g1, ([s1, t1], []))` fails, i.e., $g1$ is simulated by $([s1, t1], [])$ which proves that a choreographer exists.

Observe that, three transitions in U^T -automaton are ex-

plored

$$\begin{aligned}
 (s_1, t_1)\emptyset &\xrightarrow{SSN/CCN} [(s_2, t_1)\{SSN, CCN\}, \\
 (s_2, t_1)\{SSN, CCN\} &\xrightarrow{CCN/Appr} (s_2, t_2)\{SSN, CCN, Appr\} \\
 (s_2, t_2)\{SSN, CCN, Appr\} &\xrightarrow{CCN/Loan} (s_2, t_3)\{SSN, CCN, Appr\}
 \end{aligned}$$

instead of the entire automaton. This is because the evaluation proceeds in a goal-directed fashion and does not explore transitions that are not triggered by SSN as input from the start state of the U^T -automaton.

4 Experimental Results

We evaluate the technique using common use cases. The first example is an travel reservation service, modified from [14], where the aim is to develop a composite service (a) for reserving transportation (car-rental) from source address to the origin airport, from destination airport to destination hotel and (b) for buying the airline tickets and (c) for booking accommodation (hotel). The composition is realized from pre-existing services one each for car-rental, air-ticket purchase and hotel reservation. Note that the car-rental service is used twice in the example, once to reserve car from source address to the source airport and then to reserve car from destination airport to the hotel. The second example involves ordering, buying and shipping items, simplified from

Name	Goal Size		No. of Services	U^T size		Transitions Gen. & Expl.	Choreographer	
	States	Trans.		States	Trans		States	Trans
Travel Reservation	9	8	4	400	9659	28	15	14
Purchase & Ship	4	3	3	32	183	8	8	7
Loan 1	7	6	4	48	610	6	7	6
Loan 2	9	8	5	144	4548	8	8	8
Loan 3	11	10	6	432	26712	10	9	10

Table 1: Experimental Results.

[1]. The choreographer in this case uses individual services to do credit check, item availability check and shipping. Finally, we have also experimented with a loan approval service where the goal is approval for a specific loan amount given the credit information of the user. Once the credit check is validated, approval of specific amount may depend on approval from several different branches of loan office. We increased the number of states and transitions in the goal as well as the number of services by varying the number of branches that need to approve a loan.

For each of the above examples, Table 1 presents the size of the goal, number of existing services involved in realizing the goal, size of the U^T -automaton, number of transitions explored by our local and on-the-fly algorithm and the size of the choreographer generated. The table shows that the number of states and transitions generated and explored is much less than the total number of states and transitions in the U^T -automaton providing strong testimony that our local and on-the-fly algorithm can be effectively applied in practical setting.

5 Related Work

A number of tools has been developed for address the problem of Web service composition ranging from providing better interfaces for manual composition to providing automated formal techniques for high-level transition-system based composition (see [2, 3, 9]).

Logic based solution to address service composition problems is not new. In [5], the authors apply the golog language based on Situation Calculus for composing services. [4] uses structural synthesis of program for composition whereas [15, 12] uses theorem provers. [11] uses Linear Logic to realize a semantic composition.

Recently, [10] provides extension of the work in [13] to compare on-the-fly and once-for-all process level composition. Here, the authors refer to on-the-fly as composing available services to serve one-shot user request on an “as needed” basis, whereas, once-for-all composition builds a composition which can serve multiple user requests.

In contrast to the existing work, which focus on the theoretical and formal methodology to solve service composition, the focus of this paper is to develop a practical and efficient implementation strategy for service composition; the encoding is based on our prior work on *i/o*-automata based composition [7]. We present a complete and direct logical encoding of service composition problem and show that goal-directed evaluation of our logical encoding results

in a local and on-the-fly algorithm for proving choreographer existence and synthesizing it, when one exists. Unlike [10] we use the terminology on-the-fly to mean construction of composition of existing services as and when needed to realize a desired composite service.

6 Conclusion

In this paper we presented a local and on-the-fly technique for verifying the existence choreographer and synthesizing it. This is achieved by logical encoding of choreographer-based composition problems and evaluating the logic program in a goal-directed fashion. The results prove that our technique is promising and can be used effectively in practical settings. One of the future avenues of research is to develop heuristics to further assist and guide local exploration of the composite services. Another interesting problem will be to take into consideration data-exchange in addition to input/output actions. Here the main challenge will be to analyze data valuations with infinite domain which may make the composition problem undecidable and therefore, identify the conditions under which the problem becomes decidable.

References

- [1] D. Berardi, D. Calvanese, G. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Conference on Very large data bases*, Trondheim, Norway, 2005.
- [2] R. Hull and J. Su. Tools for composite web services: A short overview. *ACM SIGMOD Record*, 34(2), June 2005.
- [3] J. B. K. S. May Chan and L. Baresi. Survey and comparison of planning techniques for web services composition, technical report. Technical report, February 2007.
- [4] S. Lammernann. *Runtime Service Composition via Logic-Based Program Synthesis*. Ph.D. Dissertation, Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden, 2002.
- [5] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *International Conference on the Principles of Knowledge Representation and Reasoning (KRR'02)*, 2002.
- [6] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.
- [7] S. Mitra, R. Kumar, and S. Basu. Automated choreographer synthesis for web services composition using *i/o* automata. In *IEEE International Conference on Web Services (ICWS)*, Salt Lake City, 2007.
- [8] H. Nezhad, F. Benatallah, B. Casati, and F. Toumani. Web services interoperability specifications. *Computer*, 39(5):24–32, 2006.
- [9] J. Peef. Web service composition as ai planning - a survey, technical report. Technical report, University of St. Gallen, 2005.
- [10] M. Pistore, P. Roberti, and P. Traverso. Process-level composition of executable web services: on-the-fly versus once-for-all composition. In *European Semantic Web Conference (ESWC'05)*, 2005.
- [11] J. Rao. *Semantic Web Service Composition via Logic-based Program Synthesis*, PhD Thesis, Norwegian University of Science and Technology, 2004.
- [12] J. Rao, P. Kungas, and M. Matskin. Logic-based web services composition: from service description to process model. In *IEEE International Conference on Web Services (ICWS'04)*, Washington, DC, July 2004.
- [13] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. Astro: Supporting composition and execution of web services. In *International Conference on Automated and Planning Scheduling (ICAPS'05)*, 2005.
- [14] Usecase. Web service use cases. <http://www.w3.org/TR/ws-arch-scenarios/>.
- [15] R. J. Waldinger. Web agents cooperating deductively. In *1st International Workshop on Formal Approaches to Agent-Based Systems*, pages 250–262. Springer-Verlag, 2001.
- [16] XSB. The XSB logic programming system, 2006. Available from <http://xsb.sourceforge.net>.