On the Fly Algorithm for the Service Composition Problem

Hikmat Farhat, Guillaume Feuillade

On-the-Fly Algorithm for the Service Composition Problem

Hikmat Farhat
Computer Science Department
Notre Dame University
Lebanon
Email: hfarhat@ndu.edu.lb

Guillaume Feuillade
Université Paul Sabatier
Toulouse, France
Email:guillaume.feuillade@irit.fr

***Abstract*—The behavioral service composition problem arises when no available service can achieve a target behavior. The composition problem consists in building a special service, an orchestrator, which schedules the actions of the services to produce a behavior equivalent to the target one. In this paper, services are modeled as nondeterministic transition systems and the behavior of the composition realizes the behavior of the target. We propose an algorithm that avoids the full computation of the product of services, and instead constructs the orchestrator incrementally. Unlike most algorithms where the whole state space, which is exponential in the number of services, is visited, the proposed algorithm can find a solution by visiting only the pertinent portion of the state space. We also show that this on-the-fly behavior can be paired with a heuristic to speed up the synthesis.**

I. Introduction

The Service Oriented Computing paradigm uses independent components, called web services, as building blocks for realizing complex solutions. The main challenge in this approach is *service composition*. When a client request cannot be fulfilled with an existing service, service composition is the process of combining many available services in such a way as to satisfy the request. Since almost any software module can be turned into a web service, SOC has been gaining popularity.

There are many approaches for the composition problem, ranging from model checking [1],to theorem proving [2] (see [3] for a survey). The framework we use in this paper, first proposed in [4], and usually referred to as the "Roman Model", has been dealt with in many works [5][6][7]. Most solutions, to date that are based on the this framework have either an elevated complexity (e.g. [4]) or used a global approach (e.g. [8]) in which the whole state space, which is exponential in the number of services, needs to be generated beforehand.

Our contribution is a new on-the-fly algorithm that searches part of the state space. While its worst-case complexity is also exponential in the number of services (this is a lower bound, see [9]) we argue that in the average case it is much better. This paper improves on previous approaches to the problem and advances the state of the art in service composition by proposing an algorithm that: 1) visits states as needed, which allows it to deal efficiently with systems containing a large number of complex services; 2) is self-contained and can be easily incorporated in any other model.

In section II the formal setting of the problem is presented. The local algorithm is presented and analyzed in section III. Finally, we conclude with section IV.

II. Background

In this section we give the necessary definitions and formally set the service composition problem. Before stating the problem formally, it is useful to give an overview of the involved components and how they fit in the framework. In this work we follow the line of reasoning in [8], [10], originally proposed in [4]. These needed components include: a *target service*, a set of *available services*, and an *environment*.

The environment is a system shared by all the services, which allows them to maintain state and communicate. The environment also serves as a vehicle to impose behavioral constraints on the actions of some services.

Having introduced the components we can state the service composition problem informally as: given a target service and a set of available services, find an orchestrator, if one exists, that delegates requested actions to suitably chosen available services, such that the system will have the same behavior as the target service. Next we give the formal definition of all components as well as the composition problem.

Definition 1: An environment $\mathcal{E}$ is a tuple $\mathcal{E} = \langle E, \Sigma, e^0, \delta_E \rangle$ where: E is a finite set of states, e^0 is the initial state, Σ is the set of actions that can be performed, and $\delta_E \subseteq E \times \Sigma \times E$ is the transition relation.

It is convenient to write $(e_1, a, e_2) \in \delta_E$ as $e_1 \xrightarrow{a} e_2$. Such transition means that when the environment is in state e_1 and an action a is performed it will move to a new state e_2.

Available services. The available services are a set of components that can be *partially* controlled by the orchestrator and interact with the environment. Each service is defined formally as:

Definition 2: An available service $\mathcal{S}$ over an environment $\mathcal{E}$ is a tuple $\mathcal{S} = \langle S, \Sigma, s^0, G, \delta \rangle$ where: S is a finite set of states, Σ is a finite set of actions, identical to the environment's, s^0 is the initial state, G is a set of boolean functions that are used to impose constraints on some actions: $g : E \rightarrow \{true, false\}$ where E is the set of environment states. Finally, $\delta \subset S \times G \times \Sigma \times S$ is the transition relation.

When $(s, g, a, s') \in \delta$ we write $s \xrightarrow{g,a} s'$. A service can make a transition only if the state of the environment allows

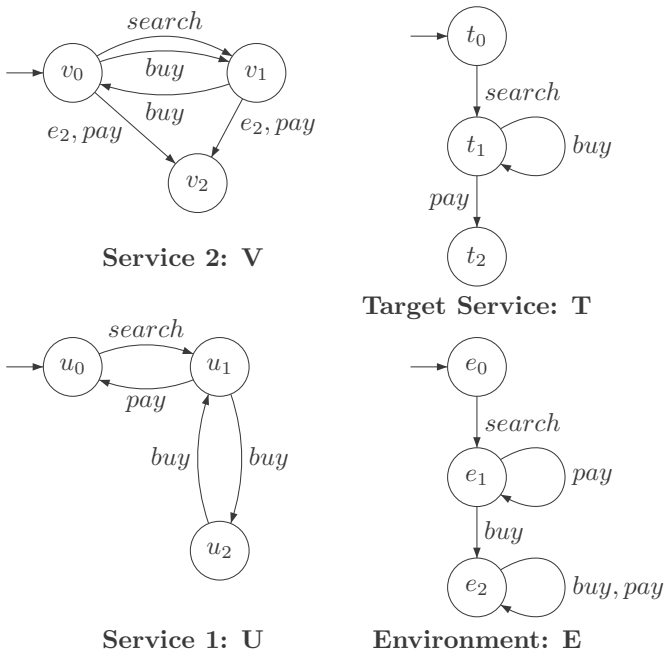

Figure 1. Example setup: U and V are available services, E the environment and T the target service.

it. So if the environment is in state e then the service can make the transition only if $g(e) = true$. Given a service $\mathcal{S} = \langle S, \Sigma, s^0, G, \delta \rangle$ and environment $\mathcal{E} = \langle E, \Sigma, e^0, \delta_E \rangle$ a *trace* of $\mathcal{S}$ on $\mathcal{E}$ is a, possibly infinite, sequence of the form $(s^0, e^0) \xrightarrow{a^1} (s^1, e^1) \xrightarrow{a^2} \dots$ where $s^i \in S$, $e^i \in E$,$a^i \Sigma$, and for all i if $(s^i, e^i) \xrightarrow{a^{i+1}} (s^{i+1}, e^{i+1})$ then $s^i \xrightarrow{g, a^{i+1}} s^{i+1}$ in $\mathcal{S}$ with $g(e^i) = true$ for some $g \in G$ and $e^i \xrightarrow{a^{i+1}} e^{i+1}$. Thus a service can make a transition only if the environment can make the same transition, and is in the appropriate state e. A *history* is a finite prefix of a trace, ending in a state. Given history $h = (s^0, e^0) \xrightarrow{a^1} \dots \xrightarrow{a^i} (s^i, e^i)$ the last state in the history is denoted by $last(h) = (s^i, e^i)$ and the length the history, denoted by $|h| = i$, is the number of actions performed. All available services together with the environment are grouped into a *community of services*, the set $\mathcal{C} = \{\mathcal{S}_1, \dots, \mathcal{S}_n, \mathcal{E}\}$.

Target service. The target service is the goal service requested by the client and which the community tries to satisfy by composing an *equivalent* service from the available services. The target service, denoted by $\mathcal{S}_t$, is defined like any other service over the same environment, except that it is deterministic.

Example. We give an example to illustrate the framework. The main motivation for the example is its simplicity. The setup, as shown in Figure 1, contains a target service, an environment and two available services. The target service is a kind of online bookstore, where one can search for and buy books. Note that the pay action is also a "logout" action so in case no item was purchased the pay action still makes sense. The environment enforces the idea that a search has to be done first. Also it is used as a data box where results of search and buy are saved. The first service U can do all three actions, search, buy and pay. Service U offers a kind of buy one and get one for half the price service, therefore a client has to buy items in pairs before paying. The second service V can search, buys, and pays for items. The service V forces at least one buy after a search because the pay transition can be executed only if the environment is in state e_2. To reach that state at least one item needs to be bought. This is handled by the $v_1 \xrightarrow{e_2, pay} v_2$ transition meaning that this transition can be done only when the environment is in state e_2, i.e. at least one item was purchased. It should be mentioned that transitions not labeled explicitly with an environment state are actually implicitly labeled by $g(e) = true$ for all $e \in E$.

Because both services have access to the environment where they can store and retrieve data, it is possible for one service to search for books, store the results in the environment, and the other service will read the result of the search from the environment to buy items.

A. Service Composition

Let $\mathcal{C} = \{\mathcal{S}_1, \dots, \mathcal{S}_n, \mathcal{E}\}$ be a community of services. A community trace is a sequence of the form $(s_1^0, \dots, s_n^0, e^0) \xrightarrow{k^1 a^1} (s_1^1, \dots, s_n^1, e^1) \xrightarrow{k^2 a^2} \dots$ such that for all $i > 0$ if $(s_1^i, \dots, s_n^i, e^i) \xrightarrow{k^{i+1} a^{i+1}} (s_1^{i+1}, \dots, s_n^{i+1}, e^{i+1})$ then

- $s^i_{k^{i+1}} \xrightarrow{g, a^{i+1}} s^{i+1}_{k^{i+1}}$ and $g(e^i) = true$ for some g.
- $e^i \xrightarrow{a^{i+1}} e^{i+1}$.
- $s_k^{i+1} = s_k^i$ for all $k \neq k^{i+1}$.

Community histories are finite prefixes of community traces, ending in a state. Given a community history h, we denote by $last(h)$ the last state in history h. The set of all histories of a community is denoted by $\mathcal{H}$.

Orchestrator. An orchestrator controls the actions of the community of services by selectively enabling and disabling their actions. Given a set of available services with history h and an action a, the orchestrator is a function $\Omega : \mathcal{H} \times \Sigma \times \{1, \dots, n\} \to \{0, 1\}$ that enables/disables service k for performing action a. For example, $\Omega(h, a, i) = 0$ means that after history h, service i is not allowed to perform action a. If $\Omega(h, a, i) = 1$ then service i is allowed to perform action a.

Definition 3: Given a sequence of actions (a trace) $\tau = a_1 \dots a_k = \tau' a_k$, the evolution of the community under the **control** of orchestrator Ω is defined inductively as

$$\mathcal{H}_{\tau,\Omega} = \bigcup_{h \in \mathcal{H}_{\tau',\Omega}} h \xrightarrow{a_k} \langle s'_1, \dots, s'_n, e' \rangle$$

where

- For every h, given that $last(h) = \langle s_1, \dots, s_n, e \rangle$, $h \xrightarrow{a_k} \langle s'_1, \dots, s'_n, e' \rangle$ is the set of histories obtained from h by appending to it all the transitions $\langle s_1, \dots, s_n \rangle \xrightarrow{a_k} \langle s'_1, \dots, s'_n \rangle$ having the properties: $\exists i$ such that $s_i \xrightarrow{g, a_k} s'_i$ with $g(e) = true$, $\Omega(h, a_k, i) = 1$ and $s_j = s'_j$ for all $j \neq i$.

- $\mathcal{H}_{\epsilon,\Omega} = \langle s_1^0, \ldots, s_n^0, e^0 \rangle$, the initial state where ϵ is the empty trace.

Definition 4: Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{E}\}$ be a community of services and $\mathcal{S}_t = \langle S_t, \Sigma, s^0, G, \delta \rangle$ be target service and Ω an orchestrator. We say that the community $\mathcal{C}$ controlled by the orchestrator Ω is a behavior composition of the target $\mathcal{S}_t$ if: for all traces $\tau = a_1 \ldots a_k$ and for all histories of the community $h \in \mathcal{H}_{\tau,\Omega}$ and target history $t^0 \xrightarrow{a_1} t^1 \ldots \xrightarrow{a_k} t^k$ Then

$$t^k \xrightarrow{a} t^{k+1} \Leftrightarrow h \xrightarrow{a} \langle s_1, \ldots, s_n \rangle$$

In other words, after any arbitrary sequence of actions τ if the target can make an a-transition then the controlled community can make the same transition. This means that the community controlled by the orchestrator mimics exactly the behavior of the target at every step.

Clearly, there are an infinite number of orchestrators for a any community and not all of them lead to behavior composition. For the remainder of this paper an orchestrator means an orchestrator that leads to behavior composition. We will show that an orchestrator exists if and only if a certain relation exists between the community and the target. We need the next definition.

Definition 5: Let $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{E} \rangle$ be a community of services and $\mathcal{S}_t = \langle S_t, s_t^0, \Sigma, \delta_t \rangle$ the target service over the same environment. We say that $\mathcal{C}$ is controllable with respect to $\mathcal{S}_t$ if there exist a relation $Z \subseteq S_t \times E \times S_1 \times \ldots S_n \times$ such that:

- $(s_t^0, s_1^0, \ldots, s_n^0, e^0) \in Z$.
- If $(s_t, s_1, \ldots, s_n, e) \in Z$ then for all $a \in \Sigma$ the following holds:
  - if $(s_t, e) \xrightarrow{a} (s_t', e')$ then $\exists k$ such that $s_k \xrightarrow{g,a} s_k'$ with $g(e) = true$ for some g and $(s_t', s_1, \ldots, s_k', \ldots, s_n, e') \in Z$.
  - for all $s_k \xrightarrow{g,a} s_k''$ such that $g(e) = true$ it is the case that $(s_t', s_1, \ldots, s_k'', \ldots, s_n, e') \in Z$.

Theorem 1: An orchestrator Ω exists such that the community controlled by Ω is a behavior composition if and only if the community is controllable with respect to the target.

The proof is shown in the appendix. It is basically by induction over the length of an arbitrary trace.

Since the union of two controllability relations is also a controllability relation then there exists a largest controllability relation, defined as the union of all controllability relations. Given a relation $R \subseteq S_t \times E \times S_1 \times \ldots \times S_n$ we define a function F over the set of relations over $S_t \times E \times S_1 \times \ldots \times S_n$ as follows:

$$F(R) = \{(t, e, p) \mid \forall a, (t, e) \xrightarrow{a} (t', e') \Rightarrow$$
$$(\exists k, p'.(p, e) \xrightarrow{ka} (p', e') \wedge (t', e', p') \in R$$

$$\wedge (p \xrightarrow{ka} p'' \Rightarrow (t', e', p'') \in R))\}$$

Where t, e are target and environment states. The state of the n services are represented collectively with p. It is easy to see that a relation R is an controllability relation iff $R = F(R)$. A typical procedure, similar to the one for classical equivalences and preorders [11], for computing the largest controllability relation would be to define the set of relations:

$$R_0 = S_t \times E \times S_1 \times \ldots \times S_n$$
$$R_{i+1} = F(R_i) \quad (1)$$

Since the transitions systems under study are finite then there exists a j such that $R_j = F(R_j)$. The largest fixed point, R_j, is the largest controllability relation one is seeking. Henceforth, this procedure for computing the largest controllability is referred to as "fixpoint". The important point to note is that one always starts with R_0, which is, being the product of all the states, exponential in the number of services. This means one always has to visit all the states in R_0, the full state space, and more importantly, process all transitions. Clearly this is an expensive operation and, as will be shown later, unnecessary. It is worth mentioning that other methods solving the composition problem, e.g. [12], also start from the full state space and remove, one by one, non matching states to obtain a solution.

We illustrate with the example discussed earlier. The full state space and a possible solution, together with the target, are shown in Figure 2. When the largest controllability is computed, the dashed state is related to the target state (t_0, e_0), the double ovals are related to the state (t_1, e_1), the ovals are related to (t_1, e_2) and *all* community states are related to (t_2, e_1) and (t_2, e_2) since these two target states don't have any transitions, and thus are related to any other state. On the other hand we show in red a much smaller controllability relation. The red transitions are the composition, or the "proof" that the relation is indeed a controllability relation. If one starts with (u_0, v_0, e_0) and at each step, and each target transition, tries to match it with a community transition it is possible to obtain an controllability relation in a much smaller number of computations than required by a fixpoint algorithm. One can see that in this case a fraction of the state space is visited whereas in the fixpoint algorithms all the state space, which includes testing all the transitions, will be visited. The solution in red can be obtained, for example, if the user has a preference for service 1, which means that the system will always try to match the target transition using service 1. Alternatively, one can use information obtained from a different method, say some kind of abstraction, to determine that among the two possible s transitions from (u_0, v_0, e_0), choose (u_0, v_0, e_0), and therefore prune the search space. Obviously all the aforementioned techniques are just heuristics and it is possible that such on-the-fly procedure makes the "worst" decision on every step.

III. On-the-Fly Algorithm

In this section we present an on-the-fly algorithm to find a controllability relation, if one exists. Let S_i be the set of states of available service i, E be the set of environment states and S_t the set of states of the target service. The algorithm maintains two relations $\mathcal{A}$ and $\mathcal{B}$, both initially empty.

The relation $\mathcal{A} \subseteq S_t \times E \times S_1 \times \ldots \times S_n$, represents the controllability relation that the algorithm is trying to find

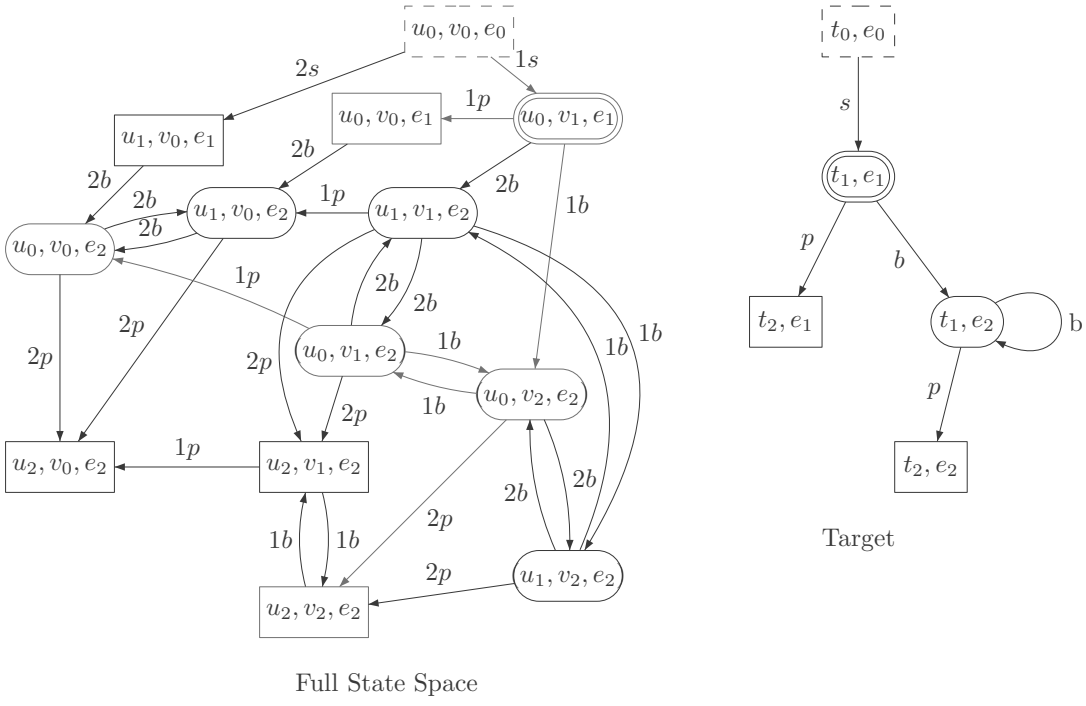

Figure 2. Full state space of the community and the target service. The red nodes and transition denote one controllability relation. The transitions search, pay, and buy are shortened to s,p, and b respectively.

between the states of the community and the target. Note that there might be more than one controllability relations.

During the execution of the algorithm states are added and *removed* from $\mathcal{A}$. The second relation, $\mathcal{B} \subseteq S_t \times E \times S_1 \times \ldots \times S_n$ represents the set of states that were found by the algorithm to be not related. Because two states found to be not related cannot become related at some later stage, states are added to $\mathcal{B}$ but never removed. The set $\mathcal{B}$ is maintained so that a given state is not processed more than once. The algorithm is composed of two mutually recursive functions $CONTROL$ and $MATCH$ that are described next.

Function CONTROL. Given a target state (t, e), and a community state $(s_1, \ldots, s_n, e)$ the function $CONTROL(t, e, s_1, \ldots, s_n)$ returns true iff the states $\langle t, e, s_1, \ldots, s_n \rangle$ are related. Basically, $CONTROL$ performs a depth-first search over the state space. When a state is visited for the first time, i.e. not in $\mathcal{A}$ nor in $\mathcal{B}$, it is assumed to be related and therefore added to $\mathcal{A}$ (lines 2-6). Then the state is processed by checking that every transition of the target can be matched by a transition of the community (lines 8-17).

After a state is processed, if it is found to be not related, then it is removed from $\mathcal{A}$ and added to $\mathcal{B}$ (lines 19-21).

Given a target state (t, e) and a community state $(s_1, \ldots, s_n, e)$, the function $CONTROL$ tests whether they are related. This is the case iff for every possible transition of the target state $(t, e) \xrightarrow{a} (t', e')$ the community can match it with an "a" transition to a state that is related to (t', e') (lines 8-17 in $CONTROL$).

Function MATCH. For every target transition, this function tries to find a community transition that matches it. It is possible that there could be multiple services that can make an "a" transitions from the current state of the community. The $MATCH$ function needs to try them one by one until it finds a match. To this end, $MATCH$ maintains all potentially valid system transitions in a queue.

QUEUE. The algorithm maintains a queue that holds all potential transitions of the system from state $(s_1, \ldots, s_n, e)$ that can potentially match a target transition $(t, e) \xrightarrow{a} (t', e')$. Since for every system state $(s_1, \ldots, s_n, e)$ and target transition $(t, e) \xrightarrow{a} (t', e')$ pair we have a different set of possible transitions the algorithm maintains a different queue for each. Therefore the queue Q_s used in the $MATCH$ function, is indexed by s which is a shorthand for $s_1 \ldots s_n t e t' e'$. Note that a given Q_s is created when it is needed and keeps the state between different calls of $MATCH$. Once we find that a transition does not match we discard it and dequeue the next possible transition. We keep doing this until a matching transition is found or the queue becomes empty. If the queue becomes empty then there is no match and the function $MATCH$ returns the value 0, no matching service is found. In this work we use a FIFO queue but one can as well use a priority queue where the priority is assigned for a given service according to some user preference to implement non-functional requirements, or as a quality of service weight. Also, if the algorithm uses a heuristic based on some already obtained information that makes one transition more likely to succeed, it will be given higher weight. Finally, the queue is defined in such a way that if there are, in a given service k, many s'_k such that $s_k \xrightarrow{a} s'_k$ then $ENQUEUE(Q_s, k, s')$ will add the first such s'_k only.

```
1  CONTROL (t, e, s_1, ..., s_n)
2  if ⟨t, e, s_1, ..., s_n⟩ ∈ B then
3  |  return false
4  if ⟨t, e, s_1, ..., s_n⟩ ∈ A then
5  |  return true
6  A = A ∪ ⟨t, e, s_1, ..., s_n⟩
7  res=true
8  foreach a ∈ Σ do
9  |  foreach (t, e) --a--> (t', e') do
10 |  |  CONT: k=MATCH (s_1, ..., s_n, e --a--> e', t --a--> t')
11 |  |  if k=0 then
12 |  |  |  res=false
13 |  |  |  Goto Exit
   |  |  |  /* Now for the reverse match    */
14 |  |  |  foreach (s_1, ..., s_k, ..., s_n, e) --a-->
   |  |  |  (s_1, ..., s'_k, ..., s_n, e') do
15 |  |  |  |  res=CONTROL (t', e', s_1, ..., s'_k, ..., s_n)
16 |  |  |  |  if res=false then
   |  |  |  |  |  /* this community state did
   |  |  |  |  |     not match. Try a
   |  |  |  |  |     different possibility
   |  |  |  |  |     from the queue          */
17 |  |  |  |  |  Goto CONT
18 Exit:
19 if res=false then
20 |  B = B ∪ ⟨t, e, s_1, ..., s_n⟩
21 |  A = A − ⟨t, e, s_1, ..., s_n⟩
22 |  changed = true
23 return res
```

Algorithm 1: function CONTROL

Theorem 2: The algorithm $CONTROL$ is polynomial in the number of states of a given service and exponential in the number of services.

The proof is shown in the appendix. The algorithm is essentially a depth-first search (DFS) on a graph where each node is visited only once. A similar reasoning to the complexity of DFS leads to the result.

IV. Conclusion

We have proposed a new on-the-fly search algorithm for the service composition of partially controllable web services. The worst-case complexity of the algorithm matches the known lower bound for the problem. However, in practice it will have better performance due to its local nature. Moreover, the algorithm doesn't need to build a priori the composition state space which allows it to handle larger system than solutions using a global search technique.

We believe that the presented algorithm can be improved further by providing it with additional information that allows it to make better decisions when confronted with equivalent choices. One approach that we are currently working on, is to use an abstraction techniques, similar to the ones used in model checking, to gain additional information that improves the decision process of the algorithm. Another issue would be to incorporate non-functional requirements in the framework as well as quality of service, which could be done by using a priority queue as suggested in the course of the paper.

```
1  MATCH ((s_1, ..., s_n, e --a--> e', t --a--> t')
   /* Let s = s_1 ... s_n tt'ee' serve as a queue
      index. This way a separate queue Q_s
      is created for each different index
      s                                        */
   /* Also Q_s is initialized at the start
      of every run but maintains state
      within one run.                          */
2  if Q_s does not exist then
3  |  create Q_s
4  |  for i = 1 to n do
5  |  |  if s_i --g,a--> s'_i ∧ g_i(e) = true then
6  |  |  |  ENQUEUE (Q_s,i,s'_i)
7  res = false
8  while Q ≠ ∅ ∧ res = false do
9  |  s'_k = DEQUEUE(Q_s)
10 |  res=CONTROL(t', e', s_1, ..., s'_k, ..., s_n)
11 |  if res=false then
12 |  |  k = 0
13 return k
```

Algorithm 2: function MATCH

References

[1] Y. Feng, A. Veeramani, R. Kanagasabai, and S. Rho, "Automatic service composition via model checking," in *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*, 2011, pp. 477–482.

[2] P. Papapanagiotou and J. Fleuriot, "Formal verification of web services composition using linear logic and the pi-calculus," in *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, 2011, pp. 31–38.

[3] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proceedings of the First international conference on Semantic Web Services and Web Process Composition*, ser. SWSWPC'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 43–54.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella, "Automatic composition of e-services that export their behavior," in *ICSOC*, 2003, pp. 43–58.

[5] P. Balbiani, F. Cheikh, and G. Feuillade, "Composition of interactive web services based on controller synthesis," *Congress on Services - Part I, 2008. SERVICES '08. IEEE*, pp. 521–528, July 2008.

[6] G. De Giacomo and F. Patrizi, "Automated composition of nondeterministic stateful services," in *Proceedings of the 6th international conference on Web services and formal methods*, ser. WS-FM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 147–160. [Online]. Available: http://dl.acm.org/citation.cfm?id=1880906.1880915

[7] D. Berardi, F. Cheikh, G. D. Giacomo, and F. Patrizi, "Automatic service composition via simulation," *Int. J. Found. Comput. Sci.*, vol. 19, no. 2, pp. 429–451, 2008.

[8] G. D. Giacomo, F. Patrizi, and S. Sardiña, "Automatic behavior composition synthesis," *Artif. Intell.*, vol. 196, pp. 106–142, 2013.

[9] A. Muscholl and I. Walukiewicz, "A lower bound on web services composition," *Logical Methods in Computer Science*, vol. 4, no. 2, 2008.

[10] S. Sardina, F. Patrizi, and G. De Giacomo, "Behavior composition in the presence of failure," in *Proceedings of Principles of Knowledge Representation and Reasoning (KR)*, G. Brewka and J. Lang, Eds. AAAI Press, 2008, pp. 640–650.

[11] R. Cleaveland and O. Sokolsky, *Handbook of Process Algebra*. Elsevier, 2001, ch. Equivalence and Preorder Checking for Finite-State Systems, pp. 391–424.

[12] T. Ströder and M. Pagnucco, "Realising deterministic behavior from multiple non-deterministic behaviors," in *Proceedings of the 21st international jont conference on Artifical intelligence*, ser. IJCAI'09, 2009, pp. 936–941. [Online]. Available: http://dl.acm.org/citation.cfm?id=1661445.1661594

V. APPENDIX

A. Proof of theorem 1

Proof: We prove the theorem by induction on the length of the trace $\tau = a_1 \dots a_k$. Assume that R is a controllability relation between community and target. It is convenient to refer to a community state with a single symbol so let $s^m = \langle s_1^m, \dots, s_n^m \rangle$.

Base case: Since R is a controllability relation then if $t^0 \xrightarrow{a_1} t^1$ then $\exists k$ such that $s_k^0 \xrightarrow{a_1} s_k^1$ and $(t^1, s^1) \in R$. Choose $\Omega(s^0, k, a_1) = 1$ and $\Omega(s^0, i, a_1) = 0$ for all $i \neq k$. Since a_1 is arbitrary then $t^0 \xrightarrow{a_1} t^1 \Leftrightarrow h \xrightarrow{a_1} s^1$.

Induction hypothesis: assume that $t^{l-1} \xrightarrow{a^l} t^l \Leftrightarrow h \xrightarrow{a^l} s^l$ for all histories h of length l. Furthermore, $(t^l, s^l) \in R$.

Induction step: since $(t^l, s^l) \in R$ then $t^l \xrightarrow{a^{l+1}} t^{l+1}$ implies that $\exists k$ such that $s_k^l \xrightarrow{a^{l+1}} s_k^{l+1}$ and thus $s^l \xrightarrow{a^{l+1}} s^{l+1}$ then choose $\Omega(h, k, a^{l+1}) = 1$ and $\Omega(h, i, a^{l+1}) = 0$ for all $i \neq k$. ■

B. Correctness and Complexity of the Algorithm

Let n be the number of available services with each service having N_i states, N_t the number of target service states, and N_e the number of environment states. Let $N = N_t \times N_e \times N_1 \times \dots \times N_n$.

Theorem 3: The algorithm $CONTROL$ terminates in a finite number of steps and when it does it returns $true$ iff $(t^0, e^0, s_1^0, \dots, s_n^0)$ are related.

Proof: First we prove the termination. Let $CONTROL_i$ be the i^{th} iteration of $CONTROL(t^0, e^0, s_0^0, \dots, s_n^0)$ and $\mathcal{B}_i$ the set of states that are not related after $CONTROL_i$ finishes. The variable $changed$ is set to $true$ iff during the run $\exists (t, e, s_1, \dots, s_n) \notin \mathcal{B}_{i-1}$ and $(t, e, s_1, \dots, s_n) \in \mathcal{B}_i$, meaning that $(t, e, s_1, \dots, s_n)$ was found to be not related during the execution of $CONTROL_i$. Recall that at no point in the algorithm, states are removed from $\mathcal{B}$. But if no new state is added to $\mathcal{B}$ then the algorithm stops. This means the set $\mathcal{B}$ is strictly increasing. On the other hand, the total number of states N is finite. Then there is an iteration j such that the variable $change = false$ and at that point the algorithm terminates.

Next we show that it yields the correct result. Observe that in a given iteration i of the algorithm, we have that if $CONTROL(t, e, s_1, \dots, s_n)$ returns $true$ it means that it has finished processing the state $\langle t, e, s_1, \dots, s_n \rangle$ and that $\langle t, e, s_1, \dots, s_n \rangle \in \mathcal{A}$. Also, recall that it returns $true$ iff for every a:

1) And for every transition $(t, e) \xrightarrow{a} (t', e')$ there exists a community transition $(s_1, \dots, s_k, \dots, s_n, e) \xrightarrow{a} (s_1, \dots, s_k', \dots, s_n, e')$ such that $\langle t', e', s_1, \dots, s_k', \dots, s_n \rangle \in \mathcal{A}$.
2) And for every s_k'' such that $(s_1, \dots, s_k, \dots, s_n, e) \xrightarrow{a} (s_1, \dots, s_k'', \dots, s_n, e')$ it is the case that $\langle t, e, s_1, \dots, s_k'', \dots, s_n \rangle \in \mathcal{A}$.

The above two conditions hold in the final iteration, when $changed = false$ and therefore no $\langle t', e', s_1, \dots, s_k', \dots, s_n \rangle$ was removed from $\mathcal{A}$, imply that the relation $\mathcal{A}$ is a controllability relation.

■

Theorem 4: The algorithm $CONTROL$ is polynomial in the number of states of a given service and exponential in the number of services.

First recall that $N = N_t \times N_e \times N_1 \times N_2 \times \dots \times N_n$ is the number of possible states of the community and target combined. Since we are doing a worst-case analysis, we assume that all the above states are reachable.

In a single run of $CONTROL(t^0, e^0, s_1^0, \dots, s_n^0)$ each state is considered once. This is because after the first visit it is either in $\mathcal{A}$ or in $\mathcal{B}$. On any subsequent call it will not be visited again (lines 2-5 in $CONTROL$). This means that each iteration of $CONTROL(t^0, e^0, s_1^0, \dots, s_n^0)$ considers at most N states. Next we compute the cost of visiting a single state. The loops in lines 8-17 have the following cost:

$$\sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |MATCH(s_1, \dots, s_n, e \xrightarrow{a} e', t \xrightarrow{a} t')|$$
$$= \sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}|$$

The last equality is true because for a given $(t, e) \xrightarrow{a} (t', e')$ the function $MATCH$ will process at most $|\{(s_1, \dots, s_n, e) \xrightarrow{a}\}|$ transitions. The above is the contribution of a single state. Because every state is visited at most once the total cost of one iteration of $CONTROL$ is

$$= \sum_e \sum_t \sum_{s_1, \dots, s_n} \sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}|$$
$$\leq \left(\sum_t \sum_e |\{(t, e) \xrightarrow{a}\}| \right) \cdot \left(\sum_{s_1, \dots, s_n} \sum_a |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}| \right)$$
$$= |L_t| \cdot |L_s| \quad (2)$$

Where L_t is the number of transitions of the target system synchronized with the environment and L_s is the number of transitions of the asynchronous product of all the services, synchronized with the environment. To get an idea about the complexity of the algorithm as a function of the number of services, n, we note that for a given action a, if service i can make $|L_{ai}|$ transitions then the asynchronous product can make $\prod_i |L_{ia}|$. On the other hand, the system cannot make a transition unless the environment does so then we get:

$$|L_s| = \sum_a |L_{ea}| \cdot |L_{1a}| \cdots |L_{an}|$$

In the worst-case every state has an "a" transition to every other state. Thus $|L_{ia}| = O(N_i^2)$, where N_i is the number of states in service i. Finally, the complexity of processing a *single* iteration of $CONTROl$ is

$$O(N_t^2 \cdot N_e^2 \cdot N_1^2 \cdots N_n^2)$$

Since $CONTROL$ is called at most $O(N_t \cdot N_e \cdot N_1 \cdots N_n)$ times on $(t^0, e^0, s_1^0, \dots, s_n^0)$, the total complexity is $O(N_t^3 \cdot N_e^3 \cdot N_1^3 \cdots N_n^3)$. Therefore, the algorithm is polynomial in the number of states of target, environment, or a given services. It is exponential in the number of services. Considering that the problem is EXPTIME-hard [9], this is optimal.