Protocol-Based Web Service Composition

Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani

LIMOS - CNRS UMR 6158
Universit Blaise Pascal, Clermont-Ferrand
{ragab,nourine,ftoumani}@isima.fr

Abstract. We study the problem of web service protocol composition. We consider a formal framework where service business protocols are described by means of Finite State Machines (FSM) and focus on the protocol synthesis problem, i.e., how to generate automatically a new target service protocol by reusing some existing ones. We consider a general case of this problem where the number of instances of existing services that can be used in a given composition is not bounded *a priori*. We motivate the practical interest of investigating such a problem and then we prove its decidability by providing a sound and complete composition algorithm. Since the main composition algorithm is not primitive recursive, which means that no theoretical complexity bound can be computed, we evaluated experimentally the performance of the algorithm on synthetic data instances and present preliminary results in this paper.

1 Introduction

Web services is an emerging computing paradigm that tends to become the dominant technology for interoperation among autonomous and distributed applications in the Internet environment [1]. Informally, a *service* is a self-contained and platform-independent application (i.e., program) that can be described, published, and invoked over the network by using standards network technologies. One of the ultimate goals of the web service technology is to enable rapid low-cost development and easy composition of distributed applications, a goal that has a long history strewn with only partial successes. To achieve this goal, there has been recently numerous research work [2,3,4,5,6,7,8,9,10] on the challenges associated with web service composition. The research problems involved by service composition are varied in nature and depends on several issues such as the kind of the composition process, e.g., manual v.s. automatic, the model used to describe the services, etc (e.g., see [3]). A line of demarcation between existing works in this area lies in the nature of the composition process: manual v.s. automatic. The first category of work deals generally with low-level programming details and implementation issues (e.g., WS-BPEL[1]) while automatic service composition focuses on different issues such as composition verification [2,7,8], planning [9,10] or synthesis [4,5,6].

[1] http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

A. Bouguettaya, I. Krueger, and T. Margaria (Eds.): ICSOC 2008, LNCS 5364, pp. 38–53, 2008.

In this paper we investigate the problem of automatic web service composition. We consider more particularly the composition synthesis problem, i.e., how to generate automatically a new target service by reusing some existing ones. We consider this problem at the web service business protocol abstraction level. A web service business protocol (or simply, a service protocol) is used to describe the external behavior of a service. Recent works have drawn attention to the importance of the state machine based formalisms for modeling the external behaviors of web services [4,11]. Continuing with this line of research, we build our work upon a formal framework where web service business protocols are described by means of Finite State Machines (FSM) and we concentrate on the following protocol synthesis problem: *given a set of n available web service protocols $P_1, \ldots, P_n$ and a new target protocol P_T, can the behavior described by P_T be synthesized by combining (parts of) the behaviors described by the available protocols.* This problem has already been addressed in recent literature [4,5,6] under the restriction that the number of instances of an existing service that can be used in a composition is bounded and fixed *a priori*. We call this restricted form of the composition problem *the bounded instances protocol (composition) synthesis problem.* It should be noted that this restricted setting is not realistic and has severe practical limitations that may impede the usage of automatic service composition by organizations. Indeed, as illustrated in section 3 of this paper, some very simple cases of web service composition cannot be solved in such a restricted setting.

Contributions. In this paper, we focus on the general case of protocol synthesis problem by relaxing the restriction on the number of protocol instances that can be used in a given service composition (i.e., we consider the unbounded instances case). The decision problem underlying composition existence in such an unrestricted setting is still an open problem. This paper makes the following contributions: (i) we show that the composition existence problem can be formalized as that of checking simulation between an FSM and a **product closure** of a FSM (i.e., an iteratively infinite product of FSMs), (ii) we propose a suitable model, called Product Closure State Machine (PCSM), to describe product closure of a FSM as an infinite state machine, (iii) we extend existing works in formal models area to prove the decidability of testing simulation between a FSM and a PCSM, and (iv) we provide a sound and complete web service composition algorithm and present first experimental results regarding performance evaluation of this algorithm.

Organization. The remainder of the paper is organized as follows. Section 2 introduce basic notions. Section 3 defines the service composition problem dealt with in this paper and points out the main theoretical and practical limitations of current state of the art. Section 4 presents the main technical results: proof of the decidability of the considered composition problem as well as a protocol-based web service composition algorithm. Section 5 describes first experimental results and Section 6 concludes and draws some directions for future works.

2 Preliminaries

We recall some basic notions that will be useful for the rest of this paper. A **State Machine** M is a tuple $M =< \Sigma_M, S_M, F_M, q^0_M, \delta_M >$, where Σ_M is a finite alphabet, S_M is a set of states, $\delta_M \subseteq S_M \times \Sigma_M \times S_M$ is a set of labeled transitions (actions), $F_M \subseteq S_M$ is the set of final states and $q^0_M \in S_M$ is the initial state. If S_M is finite then M is called a Finite State Machine (FSM).

We define below the notions of intermediate and hybrid states of an FSM M which will be useful in the remainder of this paper. Let $M =< \Sigma_M, S_M, F_M, q^0_M, \delta_M >$ be a FSM. Then: (i) the set of *hybrid states* of M, denoted $H_s(M)$, contains all the final states of M that have at least one outgoing transition, and (ii) the set of *intermediate states* of M, denoted $I_s(M)$, contains the states of $S_M \setminus F_M$ that have at least one incoming and one outgoing transitions. For example, the hybrid states of the FSM P_1 depicted at figure 1(a) are $H_s(P_1) = \{VehicleSelected\}$ while the intermediate states the FSM P_2 depicted at figure 1(b) are $I_s(P_2) = \{PaymentEstimation\}$.

We provide below a definition of the notion of simulation relation between state machines. Let $M =< \Sigma_M, S_M, F_M, q^0_M, \delta_M >$ and $M' =< \Sigma_{M'}, S_{M'}, F_{M'}, q^0_{M'}, \delta_{M'} >$ be two state machines. A state $q_1 \in S_M$ is simulated by a state $q'_1 \in S_{M'}$, noted $q_1 \preceq q'_1$, iff: (i)$\forall a \in \Sigma_M$ and $\forall q_2 \in S_M$ s.t. $(q_1, a, q_2) \in \delta_M$ there is $(q'_1, a, q'_2) \in \delta_{M'}$ s.t. $q_2 \preceq q'_2$ and (ii) if $q_1 \in F_M$, then $q'_1 \in F_{M'}$. M is simulated by M', noted $M \preceq M'$, iff $q^0_M \preceq q^0_{M'}$.

Let $M =< \Sigma_M, S_M, F_M, q^0_M, \delta_M >$ and $M' =< \Sigma_{M'}, S_{M'}, F_{M'}, q^0_{M'}, \delta_{M'} >$ be two FSMs. The asynchronous **product** (or simply, product) of M and M', denoted $M \times M'$, is a FSM $< \Sigma_M \cup \Sigma_{M'}, S_M \times S_{M'}, F_M \times F_{M'}, (q^0_M, q^0_{M'}), \lambda >$ where the transition function λ is defined as follows: $\lambda = \{((q, q'), a, (q_1, {q_1}')) : ((q, a, q_1) \in \delta_M \text{ and } q' = q'_1) \text{ or } ((q', a, {q_1}') \in \delta_{M'} \text{ and } q = q_1)\}$.

Let $k > 0$ be a positive integer. The k-iterated product of a state machine M is defined by $M^{\otimes_k} = M^{\otimes_{k-1}} \times M$ with $M^{\otimes_1} = M$. Recall that a state of $M^{\otimes_k}$ is given by a tuple over $(S_M)^k$. A **product closure** of an FSM M, noted $M^{\otimes}$, is defined as follows: $M^{\otimes} = \bigcup_{i=0}^{+\infty} M^{\otimes_i}$ It is worth noting that for any finite positive integer k, the k-iterated product $M^{\otimes_k}$ is still an FSM. However, this property does not hold for $M^{\otimes}$ since product closure leads to a state machine with an infinite number of states.

Let $\mathcal{R} = \{P_1, \ldots, P_n\}$ be a set of FSMs. In the sequel we use $\odot(\mathcal{R})$ to denote the union of the asynchronous product of all the subsets elements of $\mathcal{R}$, i.e., $\odot(\mathcal{R}) = \bigcup_{\{P_{i_1}, \ldots, P_{i_n}\} \subseteq 2^{\mathcal{R}}} (P_{i_1} \times \ldots \times P_{i_n})$.

3 Web Services Composition

In this section we first define the service composition problem dealt with in this paper and then we point out the main theoretical and practical limitations in current state of the art.

Web Services Protocol Model. We consider web services described by means of their protocols. The main goal of a web service protocol is to describe the

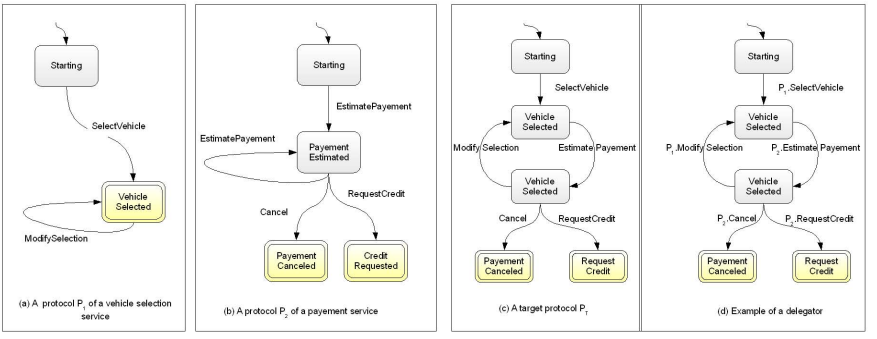

Fig. 1. An example of protocol composition

ordering constraints that govern messages exchanges between a service and its clients (i.e., messages choreography constraints). In this paper, we use the traditional deterministic finite state-machine formalism to represent messages choreography constraints (the protocol). States represent the different phases that a service may go through during its interaction with a requester. Transitions are triggered by messages sent by the requester to the provider or vice versa. Each transition is labeled with a message name. Usually the message names are followed by message polarity [11] to denote whether the message is incoming (e.g., the plus sign) or outgoing (e.g., the minus sign). For simplicity reasons, and w.l.o.g., we do not consider message polarities in this paper (i.e., incoming and outgoing messages are considered to be distinguished activities). Therefore, we obtain a web service protocol model equal the so-called *Roman model* [4], i.e., a FSM where transitions are labeled by "abstract" activities. For instance, figure 1(b) depicts the protocol of an hypothetical financing web service. The protocol specifies that the financing service is initially in the Start state, and that clients begin using the service by executing the activity estimate payment, upon which the service moves to the Payment Estimated state (transition EstimatePayement). In the figure, the initial state is indicated by an unlabeled entering arrow without source while final (accepting) states are double-circled.

The Protocol Synthesis Problem. Let us now turn our attention to the web service composition problem. We first illustrate this problem on an example. We assume a repository of two available services S_1 and S_2, respectively, described by their protocols P_1 and P_2 depicted at figure 1(a) and (b). We consider the development of a new web service S_T whose protocol P_T, called a target protocol, is depicted at figure 1(c). An interesting question is to see whether or not it is possible to implement the service S_T by combining the functionality provided by the available services S_1 and S_2. Dealing with this composition problem at the business protocol abstraction level, leads to the following question: is it possible to generate the protocol P_T by combining (parts of) the available protocols P_1 and P_2. In our illustrative case the answer is yes and an example of the composition of the target protocol P_T using the protocols P_1 and P_2 is

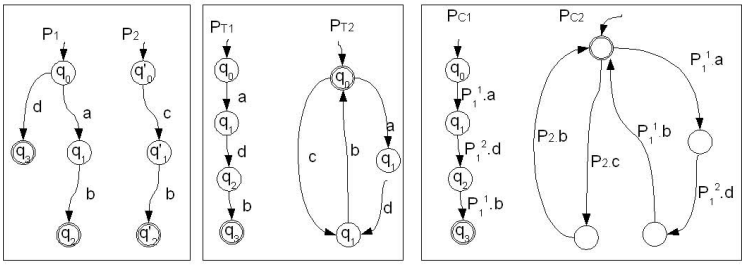

Fig. 2. Example of the composition bounded case

depicted at figure 1(d). In this case, P_T is called the target protocol while P_1 and P_2 are called the component protocols. The service composition (or *protocol synthesis*) problem is defined in [4] as the problem of generating a *delegator* of a target service using available services. A delegator is a FSM where the activities are annotated with suitable *delegations* in order to specify to which component each activity of the target service is delegated. Continuing with our example, figure 1(d) shows a *delegator* that enables to *compose* the protocol P_T using the available protocols P_1 and P_2 of figure 1(a) and ((b). For instance, this delagator specifies that the activity selectVehicle of the target protocol is delegated to the protocol P_1 while the activity estimatePayment is delegated to the protocol P_2.

The notion of a delegator is defined formally in [4] and the composition synthesis problem is expressed as the problem of finding a *"correct"* delegator for a given target protocol using a set of available protocols. A crucial question regarding this problem lies in the number of instances of the available services that can be used in a composition (i.e., to build a delegator). Figure 2 shows two examples of delegators, namely P_{C1} and P_{C2}, that use several instances of available services to respectively compose target protocols. More precisely, the delegator P_{C1} uses two instances of the protocol P_1, namely P_1^1 and P_1^2, to compose the target protocol P_{T1}. The delegator P_{C2} uses however (may be infinitely) many instances of the protocols P_1 and P_2 to compose the protocol P_{T2}. Indeed, each execution of the loop *a.d.b* (respectively, *c.b*) of the target protocol P_{T2} is realized by two new instances of the available protocol P_1 (respectively, one new instance of P_2). Hence, the number of instances of P_1, respectively P_2, that can be used to compose P_{T2} is unbounded and hence cannot be fixed *a priori*.

We provide below a definition of a generic protocol synthesis problem that makes explicit the number of instances of protocols allowed in a composition. Let $\mathcal{R}$ be a repository of services protocols, i.e., $\mathcal{R} = \{P_i,\ i \in [1,n]\}$, where each $P_i =< \Sigma_i, S_i, F_i, s_i^0, \delta_i >$ is a protocol. For each $P_i \in \mathcal{R}$, we denote by P_i^j the j^{th} instance of the protocol P_i. Given a protocol repository $\mathcal{R}$, we note by $\mathcal{R}^m = \bigcup_{i=1}^{n} \{P_i^1, \ldots, P_i^m\}$, with $m \in \mathbb{N}$.

Definition 1. ***generic protocol composition problem*** *Let $\mathcal{R}$ be a set of available service protocols and P_T be a target protocol and let $k \in \mathbb{N}$. A (generic) protocol synthesis problem, noted Compose$(\mathcal{R}, S_T, k)$ is the problem of deciding whether there exist a composition of P_T using $\mathcal{R}^k$.*

Note that, instances of this generic composition problem are characterized by the maximal number of instances of component protocols that are allowed to be used in a given composition. We distinguish in the following between two main cases, namely the bounded instances and the unbounded instances ones.

Protocol Synthesis Problem: The Bounded Case. Existing works [4,5,6] that investigated the protocol synthesis problem make the simplifying assumption that k, the number of instances of a service that can be involved in the composition of a target service is bounded and fixed *a priori*, i.e., they address the problem $Compose(\mathcal{R}, S_T, k)$. Note that this particular case, called the *bounded instance protocol synthesis problem*, can be reduced w.l.o.g to the simplest case where $k = 1$. Indeed, if $k > 1$ the problem $Compose(\mathcal{R}, S_T, k)$ can be straightforwardly reduced to the problem $Compose(\mathcal{R}^k, S_T, 1)$. The following proposition gives a formalization of the bounded protocol synthesis problem using the k-iterated product operator.

Proposition 1. *Let $Compose(\mathcal{R}, S_T, k)$ be a protocol synthesis problem with k a finite positive integer. The problem $Compose(\mathcal{R}, S_T, k)$ has a solution iff $S_T \preceq \odot(\mathcal{R}^k)$.*

The work of [4] shows that the problem $Compose(\mathcal{R}, S_T, 1)$ can be reduced to that of testing the satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic (DPDL). In [5], the PDL-based framework proposed in [4] is extended to deal with a more expressive protocol model. Interestingly, in [6] the protocol synthesis problem is reduced to the problem of testing a simulation relation between the target protocol and the product of the existing protocols. Using such a reduction, [6] shows the EXPTIME completeness of the bounded instances protocol synthesis problem.

It is worth noting that the setting of bounded instances is very restrictive in the sense that some simple protocol synthesis problems, in which the solution may use an unbounded number of instances of component protocols, cannot be solved. As an example, the rather simple composition problem depicted at figure 2, and which consists in the synthesis of the target protocol P_{T1} using the available protocols P_1 and P_2, cannot be solved by the current state of the art approaches although a solution (i.e., the delegator P_{C2}) is not complex to construct. These strong limitations motivated our work on the unbounded instance case of the protocol synthesis problem.

Protocol Synthesis Problem: The Unbounded Case. In the remainder of this paper we study the protocol synthesis problem in the case where the number of protocol instances that can be used in a composition are not bounded *a priori* (i.e., the problem $Compose(\mathcal{R}, S_T, +\infty)$). In other words, given a repository $\mathcal{R} = \{P_1, \ldots, P_n\}$ of service protocols, we consider the generation of new composite protocols that can be obtained by an asynchronous product of any subset of protocols in $\mathcal{R}^{+\infty}$. More precisely, we consider the decision problem underlying the general protocol synthesis problem, i.e., the problem $Compose(\mathcal{R}, S_T, +\infty)$.

Problem 1. Let $\mathcal{R}$ and S_T defined as previously. Is the problem $Compose(\mathcal{R}, S_T, +\infty)$ decidable?

One way to answer this open question is to consider the related 'simulation relation' decision problem. Indeed, $Compose(\mathcal{R}, S_T, +\infty)$ has a solution if S_T is simulated by a product of any subset elements of $\mathcal{R}^{+\infty}$ (i.e., $S_T \preceq \odot(\mathcal{R}^{+\infty})$). Such a characterization of solutions can also be expressed using the product closure operator as stated below.

Theorem 1. *The problem* $Compose(\mathcal{R}, S_T, +\infty)$ *has a solution iff* $S_T \preceq \odot(\mathcal{R}^{+\infty})$ *(or equivalently,* $S_T \preceq (\odot(\mathcal{R}))^{\otimes}$*).*

The main difficulty here comes from the fact that a product closure of an FSM is not an FSM. We shall prove in next section that checking simulation between an FSM M and a product closure of M (i.e., $M^{\otimes}$) is decidable. This enables to derive the decidability of the protocol synthesis problem.

4 Decidability Problem and Composition Algorithm

In this section we are interested by the problem of testing the existence of a simulation relation between an FSM and a product closure of an FSM. To investigate this problem, we need first to define a suitable state machine model that enables to describe a product closure of an FSM. Various state machine-based representations may be suitable to tackle our problem such as, for example, shuffle automata, introduced in [12] to recognize the so-called shuffle languages, or Petri Nets. However, as we deal only with a specific form of shuffle automata, i.e., automaton of the form $M^{\otimes}$ where M is an FSM, we use in our work a simpler tool. We introduce below a state machine, a *PCSM* (Product Closure State Machine), that enables to describe the product closure of an FSM. More precisely, a PCSM is an infinite state machine that describes: (i) all possible executions of a product closure of an FSM, and *(ii)* the branching choices at each state of the execution of such an state machine. It should be noted that PCSMs are a particular form of the so-called Vector Addition Systems (VAD) [14], which are nothing other than a variant mathematical notation of Petri Nets. The PCSM notation turned out to be more convenient to handle proofs and complexity analysis in our context.

Informally, the product closure $M^{\otimes}$ enables to run an infinite number of parallel instances of M. A product closure $M^{\otimes}$ may then be described by an FSM similar to M with an unbounded stack of tokens in each state. The tokens number of a stack describes the number of parallel instances having reached that state. Let $w \in \Sigma_M^*$ the input of $M^{\otimes}$, a symbol $a \in w$ is recognized by the execution of such a state machine in two cases : (i) creation of a new instance of M: if there is an outgoing transition labeled a from the initial state of M to a state q. Upon such a transition, a token is added to q, or (ii) moving an existing instance of M: if there exists two states q and q' such that $(q, a, q') \in \delta_M$ and q has one or more tokens, then upon this transition, a token is moved from q to q'.

Unlike finite state machines, where the *instantaneous description (ID)* of a given state machine is given by its current state, an ID of a PCSM involves the set of its states as well as the number of tokens in each state (number of instances having reached that state when recognizing a word). We introduce below the notion of configuration that enables to capture an ID of a PCSM. Let $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ be an FSM and let $|I_s(M)| = l$ and $|H_s(M)| = n$ be respectively the set of intermediate and hybrid states of M. We assume states of $I_s(M)$ (respectively, $H_s(M)$) ordered according to the lexicographical order and relabeled accordingly with integers from 1 to l (respectively, from $l+1$ to $l+n$). The configurations of $M^{\otimes}$ are formally defined below.

Definition 2. ***(Configuration)*** *A configuration C of the product closure $M^{\otimes}$ is a tuple of size $l+n$ of positive integers. The i^{th} element of C, written $C[i]$, denotes the number of tokens (i.e., instance of M) that are at state i. We say that $C[i]$ is the witness of the state i in a configuration C. Note that, if $i \leq l$ (respectively, $i > l$) then $C[i]$ is a witness of an intermediate state (respectively, an hybrid state).*

A configuration C is an initial (respectively, final) configuration of $M^{\otimes}$ iff $C[i] = 0$, $\forall i \in [1, l+n]$ (respectively, iff $C[i] = 0$, $\forall i \in [1, l]$.

Note that, a configuration keeps only the information about intermediate and hybrid states. Indeed, it is useless to store information about the number of tokens (i.e., instances of M) that are in final, not hybrid, states since such instances can no longer contribute to the realization of the target service. In the same spirit, as the number of instance of M that can be created is infinite (i.e., the set of tokens in the initial state is infinite) we do not describe the initial state in a configuration unless it is also an intermediate state.

Continuing with the example of figure 3, the FSM M contains only one intermediate state (state q_1) and one hybrid state (state q_2). Hence, a configuration associated with $M^{\otimes}$ is a pair of integers where the first (respectively, the second) integer is the witness of the state q_1 (respectively, q_2). For instance, a configuration $C = (2, 3)$ indicates an instantaneous description of $M^{\otimes}$ in which there are two instances of M at state q_1 and three instances at state q_2.

Using the notion of configuration, we formally define below PCSMs.

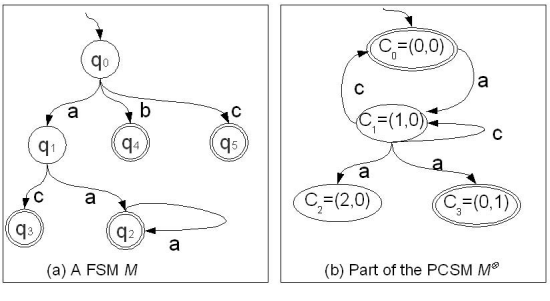

Fig. 3. An FSM and its corresponding PCSM

Definition 3. ***(PCSM)*** *Let $M =< \Sigma_M, S_M, F_M, q_M^0, \delta_M >$ be a FSM with $|I_s(M)| = l$ and $|H_s(M)| = n$. The associated PCSM of M is an infinite state machine $M^{\otimes} =< \Sigma_M, \mathcal{C}, F_C, C_0, \phi >$, where:*

- *$\mathcal{C}$ is an (infinite) set of states consisting of all the configurations of $M^{\otimes}$,*
- *F_C is the set of final configurations of $M^{\otimes}$, i.e., $\{C \in \mathcal{C} | C[i] = 0, \forall i \in [1, l]\}$,*
- *C_0 is the initial state of $M^{\otimes}$ and corresponds to the initial configuration, i.e., $C_0[i] = 0$, $\forall i \in [1, l + n]$,*
- *$\phi \subseteq \mathcal{C} \times \Sigma_M \times \mathcal{C}$ is an infinite set of transitions. The set ϕ is built as follows. Let C_1 and C_2 be two configurations in $\mathcal{C}$. We have $(C_1, a, C_2) \in \phi$ iff $(q, a, q') \in \delta_M$ and one of the following conditions holds:*
  - *$q = q_M^0$ and $q' \in (F_M \backslash H_s(M))$ with $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$, or*
  - *$q = q_M^0$ and $q' \in (I_s(M) \cup H_s(M))$ with $C_2[q'] = C_1[q'] + 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$ and $i \neq q'$, or*
  - *$\{q, q'\} \subseteq (I_s(M) \cup H_s(M))$ with $C_1[q] > 0$, $C_2[q] = C_1[q] - 1$, $C_2[q'] = C_1[q'] + 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$ and $i \notin \{q, q'\}$, or*
  - *$q \in (I_s(M) \cup H_s(M))$ and $q' \in (F_M \backslash H_s(M))$ with $C_2[q] = C_1[q] - 1$, $C_1[i] = C_2[i]$, $\forall i \in [1, l + n]$ and $i \neq q$.*

Figure 3(b) describes a part of $M^{\otimes}$, the PCSM of the FSM M depicted at figure 3(a). As mentioned before, configurations of $M^{\otimes}$ are pairs (i, j) where i (respectively, j) is the witness of the state q_1 (respectively, q_2). The infinite state machine $M^{\otimes}$ is initially in the configuration $C_0 = (0, 0)$ then it can, for example, execute the activity a, upon which it moves to the configuration $C_1 = (1, 0)$. At this stage, $M^{\otimes}$ has two possibilities to execute the activity c : (i) by moving the current instance of M that is at state q_1 into the final state q_3, or (ii) by creating a new instance of M and moving it from state q_0 into the final state q_5. Note that, as the final states q_3 and q_5 are not described in configurations, case (i) make the $M^{\otimes}$ moving back to the configuration C_0 while case (ii) makes it looping on configuration C_1.

Simulation Decision Problem. We provide below the main result of this section.

Problem 2. Let A and M be two FSMs. Is it decidable whether $A \preceq M^{\otimes}$ or, equivalently, is decidable whether $q_A^0 \preceq C_0$?.

This section answers positively to this problem by providing a sound and complete algorithm that checks the existence of a simulation relation between a FSM and a product closure of a FSM.

Note that, the main difficulty to devise our algorithm comes from the fact that we have to check the existence of a simulation relation between an FSM and a PCSM, this latter one being an infinite state machine. The corner stone of our proof is to show that to check the existence of such a simulation relation we need only to explore a finite part of the corresponding PCSM. We propose an algorithm made of three main procedures : Check-Sim, Check-Candidate and Check-Cover. When checking the simulation between a given state q and a configuration C, the Check-Sim procedure will recursively generate new simulation tests by making calls to the Check-Candidate procedure for each transition (q, a, q') in A. This latter procedure enables to check if the state q' is simulated

by at least one configuration C' such that (C, a, C') is in $M^{\otimes}$. Informally speaking, the execution of the algorithm can be seen as a tree where the nodes are labeled with pairs (q, C) and correspond to the calls of the Check-Sim algorithm. As an example, figure 4(b) shows an execution of a Check-Sim between the initial state q_1 of the FSM of figure 4(a) and the initial configuration $C_0 = (0, 0)$ of the product closure of the FSM of figure 3(a).

A crucial question is then to ensure that the algorithm terminates. Observe that for each state q', the number of candidates C' generated by the Check-Candidate procedure is linear in the size of M since for any configuration C of a PCSM $M^{\otimes}$, the number of outgoing transitions is finite and bounded by the total number of transitions in M. Therefore, to ensure termination of the algorithm it remains to show that there are no infinite branches in the execution tree of the algorithm. In the simple case where A is a loop-free FSM, it is easy to see that the corresponding execution tree of the algorithm is finite since the length of the branches are bounded by the size of the maximal path in A. For the general case, a state q belonging to a loop in A may appear an unbounded many times in a branch of the execution tree of the algorithm. Such a case is illustrated on the figure 4(b) where the branch depicted in bold involves many times the state q_1 which belongs to the loop $(ab)^*$ of the FSM A. An important technical contribution of this work is to provide necessary and sufficient conditions that enable to cut such infinite branches. This is achieved by the second terminating condition of the Check-Sim (i.e., the call to the Check-Cover procedure) which is based on the following property: if a state q appears infinitely many times in a given branch then there is necessarily a sub-path in this branch from a node (q, C) to a node (q, C') such that C' is a cover of C. Interestingly, this condition characterizes the cases where a loop in A is simulated by $M^{\otimes}$. Continuing with the example of figure 4(b), the bold branch which is potentially infinite is cut at node $(q_1, (0, 1))$ since the configuration $(0, 1)$ is a cover of the configuration $(0, 0)$ which appear previously in a node $(q_1, (0, 0))$ in the same branch. Note that, to verify such a condition, the Check-Cover procedure maintains for each state q in a given branch a list, noted $L(q)$, of all the configurations C' corresponding to the nodes (q, C') of this branch. In our example, we have at node $(q_1, (0, 1))$ of the bold branch the sequence $L(q_1) = [(0, 0), (1, 0)]$.

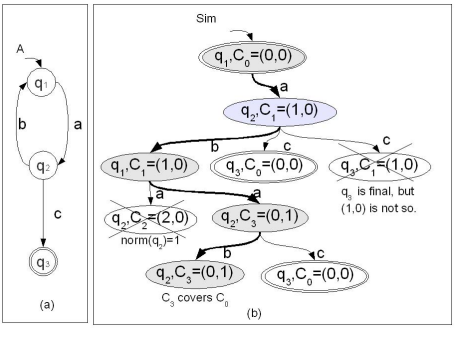

Fig. 4. Simulation of an FSM by a PCSM

Algorithm 1. Check-Sim

Input. Two FSM A and M, a state q of A, a configuration C of $M^{\otimes}$
Output. boolean
begin
  if $q \in F_A \setminus H_s(A)$ **then**
    return($\sum_{i=1}^{|I_s(M)|} C[i] = 0$);
  if *Check-Cover(q,C)* **then**
    Return(true);
  for *each transition* (q, a, q') *in* δ_A **do**
    if *not(Check-Candidate*(q', C, a)*)* **then**
      return(false);
  return(true);
end

The correctness of the algorithm Check-Sim is stated in the following theorem.

Theorem 2. *The algorithm Check-Sim halts and is sound and complete.*

The proof of this theorem, omitted here for lack of space, is available in [16]. It is worth noting that the proposed proof is constructive in the sense that if the answer is true, the algorithm may be easily modified to exhibit a simulation relation between its inputs. This is an interesting point in the context of the protocol synthesis problem since such a simulation relation can be effectively used to build a delegator.

Algorithm 2. Check-Candidate

Input. a state q' of A, a configuration C of $M^{\otimes}$, $a \in \Sigma_M$
Output. boolean
begin
  Candidates=$\emptyset$;
  for *each transition* (C, a, C') *in* ϕ **do**
    if $\sum_{i=1}^{|I_s(M)|} C'[i] \leq norme(q')$ **then**
      Candidates= Candidates $\cup\{C'\}$;
  flag=0;
  while *Candidates*$\neq \emptyset$ *and (not flag)* **do**
    C' =first element in Candidates;
    flag= Check-Sim(q',C');
  return(flag);
end

Theorem 3. *Let A and M be two FSMs. It is decidable whether $A \preceq M^{\otimes}$.*

Finally, in the following corollary, we derive the main result of this work regarding the addressed web service composition problem.4

Algorithm 3. Check-Cover

Input. a state q of A, a configuration C of $M^{\otimes}$
Output. boolean
begin
  for $C' \in L(q)$ **do**
    if $C' \triangleleft C$ **then**
      return(true);
  return(false);
end

Table 1. Description of the test sets

Test ID	#S	#M	#H	#L	Number of variants	Total number of generated tests
Test 1	200	2, 4, 8 .. 4096	c	c	12	12000
Test 2	10, 25, 50, 100, 200	2, 4, 8 .. 4096	c	c	60	60000
Test 3	100	10	from 0 to 4	c	5	5000
Test 4	100	10	c	0, 1, 2, 3	4	4000

Corollary 1. *Let* $\mathcal{R}$ *and* S_T *defined as previously. The problem* $Compose(\mathcal{R}, S_T, +\infty)$ *is decidable.*

5 Experimental Evaluation

We implemented our algorithm as part of ServiceMosaic[2], a model-driven prototype Caise tool for modeling, analyzing, and managing web services. We developped two main components: (i) WS-protocol-generator that enables to generate synthetic web service protocols according to several input parameters, such as the number of transitions per services, number of services, etc, and (ii) WS-protocol-composer an implementation of our composition algorithm. These components have been implemented using the JavaTM platform version 6 and the Eclipse framework where they are deployed as plug-ins.

Evaluation Goals. We can observe that the time complexity of our composition algorithm depends on the size of the execution tree of the algorithm Check-sim. The sizes of such a tree vary depending on two main parameters: the degrees of the nodes (i.e., the number of childrens of a given node) and the depth of the tree (i.e. the sizes of the paths between the root and the leaves).

To better understand this issue, we focused our first experiments on the analysis of the impact of the following parameters on the execution time of the algorithm:

[2] `http://servicemosaic.isima.fr`

- Number of services in the service repository, noted #S,
- Total number of distinct message labels that appear in the services of the repository, noted #M,
- Number of hybrid states in each service in the repository, noted #H,
- Level of nested loop in each service in the repository, noted #L.

Indeed, the degree of a node depends on the number of candidates computed by the procedure Check-Candidate. It corresponds to the number of transitions labeled by the same message in the services of the repository. Note that the degree does not depend on the number of active instances of a service, since using any of them leads to the same configuration. To increase the node degree one can either increase the number of services (i.e., the value of #S) or decrease the number of message labels (i.e., the value of #M). Secondly, The depth and the density of the tree depends on the presence of loops. Indeed, our proof was based on the Dickson lemma [15] which ensures the finiteness of the Check-sim procedure when hybrid states are present. This motivates the use of the parameters #H and #L in our tests. The case where only intermediate states are considered, the depth of the execution tree is bounded by a factorial function. As will be seen below, this theoretical deduction was confirmed by the results of our experimental tests.

Building the Test Sets. To achieve the aforementioned goals, we constructed 4 test sets each of which focusing on the study of some specific parameters among the ones mentioned above. Each test set describes the main features of the studied composition problem. The description of the test sets, summarized at table 1, as well as the results of the experimental evaluation are presented in the remainder of this section. The experiments have been achieved on Xeon double process HT 3GHz and 2GO of RAM. In the presented results, the execution times are given in milliseconds.

Test 1. This test set enables to assess the impact of the number of the distinct message labels that appear in the available services. For this test set (first line of the table 1, we defined a first variant with a target service and a repository of 200 available services taking their message from an alphabet of 4096 distinct labels. Then starting with this first variant, we generate other variants by relabeling at each step the messages in order to reduce the total number of distinc labels by magnetitude of 2. The total number of variants is then equal to 12 (i.e., $\#M = 4096$, $\#M = 2048$, $\#M = 1024, \ldots, \#M = 2048$). Note that, the occurrence of symbol c in the table 1 indicates a constant value, generated randomly, and used for the different variants of the test set.

For each of the variant of Test 1, we generated and runned 1000 instances. The result is reported on figure 5(a). Each point of the given curve denotes the average execution time of the 1000 instances of the corresponding variant. Observe that when $\#M$ decreases below a given threshold, namely 8 in the figure, this leads to an exponential blow up in the execution time while above this threshold, the values of the parameter $\#M$ seem to have less impact on the performance of the algorithm.

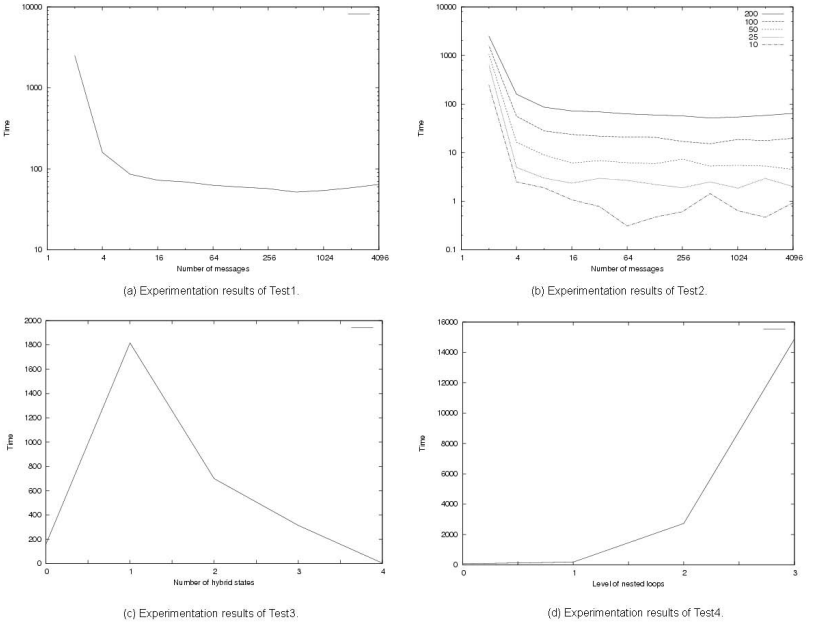

Fig. 5. Experimental evaluation results

Test 2. In addition to the number of messages labels ($\#M$), this test set enables to assess the impact of the number of services available in the service repository ($\#S$). We considered five variants of this test set obtained by varying the value of the parameter $\#S$ (respectively, 10, 25, 50, 100 and 200). As previously, for each value of $\#S$, we define several variants for different values of $\#M$ (ranging from 4096 to 2). We generated and executed 1000 instances of each variant (i.e., a total number of 60000 tests). The average execution time of each variant is reported on figure 5(b). Unsurprisingly, the results show that number of available services to explore during the composition process impacts the global performance of the algorithm. Moreover, this test set confirms the trend observed previously regarding the impact of the number of the distinct message labels on the performance.

Test 3. Test 3 studies the impact of the number $\#H$ of hybrid states (respectively, the level $\#L$ of nesting) in the available protocols. As previously, we generated a first variant of Test 3 with $\#S = 100$ and $\#M = 10$ and $\#H = 0$ (i.e., no hybrid state). Then, we generate other variants by modifying the first one by increasing the number of hybrid states (from 0 to 4). Therefore, we obtain a total number of 5 variants. We generate and executed 1000 instances of each variant. The results are depicted at figure 5(c).

Interestingly, we can observe two main phases in the results depicted on this figure. In the first phase (from $\#H = 0$ to $\#H = 1$) , the augmentation of

the number of hybrid states leads to a proportional increase of the execution time while we observe the converse behaviour in the second phase (i.e., when $\#H > 1$, the execution time decreases while $\#H$ increases). In fact, above a given threshold, adding hybrid states increases the number of accepting states making the complete conversation (i.e., accepted words) shorter.

Test 4. Test 4 studies the impact of the level $\#L$ of nested loops in the available protocols. For this test set, we generate 4 variants with a fixed set of 100 services and 10 message labels. We distinguish 4 varaiants with respect to the values of $\#L$ (ranging from 0 to 3). We executed 1000 instances of each variant and reported the average execution time in figure 5(d). As it can be expected, it turned out that the level of nesting leads to an exponential blow up in the performance of the algorithm.

6 Conclusion

We have studied the web service protocol synthesis problem in the general case where the number of protocol instances that can be used in a composition is unbounded. We made a reduction of this problem to that of of checking simulation between a FSM and a product closure of a FSM. To cope with this later problem, we first proposed PCAs as a suitable tool for describing the behavior of a product closure of an FSM and built upon this formal framework to prove the decidability of checking the simulation relation between a FSM and a PCA.

Our preliminary experimental results show that not only the total number of services in a repository, but also other parameters, such as the number of message labels or the level of the nested loops, may influence heavily the performance of the algorithm.

As a perspective of this work, we point out several interesting issues: (i) the algorithmic issues related to the optimization of the proposed algorithm as well as the development of suitable implementation strategies, (ii) complexity, by identifying particular cases that either reduce the complexity of the problem or can be solved using classical simulation algorithms, and (iii) extension of our technique to more expressive models that enable for example modeling message exchanges and impacts on the real world such as the *Colombo* model [5].

References

1. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer, Heidelberg (2004)
2. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: WWW 2003. ACM, New York (2003)
3. Dustdar, S., Schreiner, W.: A survey on web services composition. International Journal of Web and Grid Services 1(1), 1–30 (2005)
4. Berardi, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioral descriptions. IJCIS 14(4), 333–376 (2005)

5. Berardi, D., Calvanese, D., Giacomo, G.D., Hull, R., Mecella, M.: Automatic composition of transition-based semantic web services with messaging. In: VLDB, pp. 613–624 (2005)
6. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 274–286. Springer, Heidelberg (2007)
7. Narayanan, S., McIlraith, S.: Simulation, verification and automated composition of web services. In: WWW 2002, pp. 77–88 (2002)
8. Hamadi, R., Benatallah, B.: A Petri net-based model for web service composition. In: Australasian Database Conference, pp. 191–200 (2003)
9. Traverso, P., Pistore, M.: Automated Composition of Semantic Web Services into Executable Processes. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 380–394. Springer, Heidelberg (2004)
10. McIlraith, S., Son, T.: Adapting Golog for Composition of Semantic Web Services. In: KR 2002, pp. 482–493 (2002)
11. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing web service protocols. DKE 58(3), 327–357 (2006)
12. Jedrzejowicz, J., Szepietowski, A.: Shuffle languages are in p. TCS 250(1-2), 31–53 (2001)
13. Warmuth, M.K., Haussler, D.: On the complexity of iterated shuffle. JCSS 28(3), 345–358 (1984)
14. Karp, R., Miller, R.: Parallel Program Schemata. JCSS 3(2), 147–195 (1969)
15. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. Amer. Journal Math. 35, 413–422 (1913)
16. Ragab, R., Nourine, L., Toumani, F.: Web services composition is decidable, `http://www.isima.fr/ragab/RNTReport08.pdf`