International Journal of Foundations of Computer Science

AUTOMATIC SERVICE COMPOSITION VIA SIMULATION

DANIELA BERARDI[1], FAHIMA CHEIKH[2], GIUSEPPE DE GIACOMO[1], FABIO PATRIZI[1]
[1] *Dipartimento di Informatica e Sistemistica , Università di Roma "La Sapienza"*
Via Salaria 113, 00198 Roma, Italy
`lastname@dis.uniroma1.it`
[2] *Université Paul Sabatier, Institut de Recherche en Informatique de Toulouse*
31062 Toulouse Cedex 9, France
`cheikh@irit.fr`

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

In this paper we study the issue of service composition, for services that export a representation of their behavior in the form of a finite deterministic transition system. In particular, given a specification of the target service requested by the client as a finite deterministic transition system, the problem we face is how we can exploit the computations of the available services for realizing the computations of the target service. While ways to tackle such a problem are known, in this paper we present a new technique that is based on the notion of simulation, which is still optimal from the computational complexity point. Notably, such a technique, opens up the possibility of devising composition in a "just-in-time" fashion. Indeed, we show that, by exploiting simulation, it is actually possible to implicitly compute all possible compositions at once, and delay the choice of the actual composition to run-time.

1. Introduction

Service Oriented Computing (SOC) is the computing paradigm that utilizes services as fundamental elements for realizing distributed applications/solutions. Services are self-describing, platform-agnostic computational elements that are advocated to support rapid, low-cost and easy composition of loosely coupled distributed applications [2, 31, 20]. From a practical point of view services are modular applications that can be described, published, located and invoked over a network: any piece of code and any application component deployed on a system can be wrapped and transformed into a network-available service. Interestingly description of services are quite high level: typically services, or better the computations provided by the services, are described in term of finite state transition systems of some sort [17].

The availability of high level descriptions of the computations provided by a services opens the possibility of composing services in an automatic way to realize target computations. Automatic service composition has been investigated in several contexts: services seen as atomic actions, e.g., [1], by relying on the research on Planning in AI [12]; service seen as information providers, e.g., [21], by relying on the work in data integration [29, 14, 18]; and services seen as complex processes that can engage in a variety of conversations, e.g., [24, 19, 7, 5], by relying, at least implicitly, on the literature on process synthesis [25, 30, 28].

In this paper we look at the latter context. In particular we look to one of the most intriguing proposals of service composition known as the Roman Model [4, 5]. In such a proposal available services are characterized by their conversational behavior, compactly represented in terms of finite deterministic transition systems. The target of the composition is to realize a new service, specified by the client again in terms of a finite deterministic transition system, by making use of *fragments* of the computation provided by the available services.

In other words, the Roman Model envisions a kind of "service integration system". In particular the system makes available to client virtual building blocks. Making use of such virtual blocks the client can write its own service as a sort of high-level program, abstractly represented as finite deterministic transition system. In fact the virtual blocks are not implemented directly, but made available through the service composition. Indeed the actual services that are available to the system are themselves formally described in terms of high level programs that are built out of such virtual blocks. Such a description can be considered as a sort of mapping from the concrete service to the virtual blocks of the integration system. The idea is to exploit the reverse of such mapping to automatically get a realization of the virtual blocks. Obviously a service places constraints on how the virtual blocks can be used, so that the service can actually realize them. The composition must be compatible with such constraints in order to actually exist.

The main composition synthesis technique developed on the Roman Model is based on a reduction to satisfiability of a Propositional Dynamic Logic [15] formula. Such a reduction is polynomial and this gives an EXPTIME-upper-bound on the problem [4, 5]. EXPTIME-hardness of the problem was recently shown by Muscholl and Walukiewicz [23].

In this paper we look again at such form of composition, but from a very different perspective, building on the following observation: a composition exists if and only if a simulation relation [22] exists from the target to the (nondeterministic) transition system formed by the asynchronous product of the transition systems of the available services. This observation was made several times by the authors of the paper in workshop and tutorials [9, 6, 8], and it was also informally discussed in Daniela Berardi's PhD thesis [3], however it was not fully worked out in a publication yet. The connection with simulation was also independently made in [13], and although simulation is not explicitly mentioned, such connection is also related to the form treatment of to the extensions of the Roman model proposed in [11].

Once this observation is acquired we can develop a new technique for synthesizing composition that is based on computing the maximal simulation, and verify that the initial states of the target transition system and the asynchronous product of the available transition system are in the simulation. Such a computation is polynomial in the size of the target transition system and polynomial in the size of the asynchronous product of the available transition systems. As a result, the new technique is again in EXPTIME in the size of the available transition systems.

Beside these basic results, we show that synthesizing composition using simulation has a very interesting property: the maximal simulation contains enough information to allow for extracting every possible composition, through a suitable choice function. This property opens the possibility of devising composition in a "just-in-time" fashion: we compute the maximal simulation a priori, then equipped with such a simulation we start executing the composition, choosing the next step in the composition according to criteria that can depend from information that is available only at run-time (actual availability of services, network communication problems or cost, etc.). Indeed it suffices that the next step chosen for execution leads to service states that remain within the simulation relation. All in all, we believe that the synthesis technique that we propose here provides the formal basis for building compositions that are reactive, i.e., that are able to react to events that may occur at run-time.

The rest of the paper is organized as follows. First, in Section 2 and 3 we recall the notions of services and composition originally presented in [4, 5]. In Section 4 we show how simulation can be used to check for the existence of composition in an optimal way from the computational complexity point of view. Then, in Section 5, we investigate the possibility of using simulation for actually synthesizing compositions, and we show how it can be used as a sort of precomputation that allows for generating composition in a "just-in-time" fashion at run-time. Section 6, concludes the paper with some brief final remarks on the significance of the results both in the context of service compositions and in the context of simulation. In particular, wrt simulation, the results presented here close a long standing open problem.

2. Services as Transition Systems

In this section, we present the basic framework of our approach, starting from the description of services as finite *transition systems* (TSs). Besides this, further notions, which indeed characterize our approach, are introduced in order to formalize the intuitions exploited in the synthesis technique. The following paragraphs are aimed at providing, for each notion, a detailed description of such formalization along with the ideas behind it.

Services Intuitively, a service is a software artifact characterized by its behavior, that is, the potential evolutions resulting from the interaction with some external system such as, for instance, a client service. Basically, a *service* is a program intended to interact with a client. Being a service, interactions are expected to be conformant with its behavior, that is, each state defines both the allowed actions and the consequent transitions. More precisely, at each step, *(i)* the program presents the client a choice of available actions, according to its current state, *(ii)* the client instructs the program to execute one of them, *(iii)* the program executes it, moves to successor state and goes back to *(i)*. Client-service interactions can be stopped whenever the service is in a “final” state.

Since our technique aims at combining services in order to produce a desired behavior, a formal description of a service behavior is needed. In this paper, a service (behavior) is represented by a finite *deterministic* transition system $TS = \langle A, S, s^0, \delta, F\rangle$, where:

- A is the finite alphabet of actions
- S is the finite set of states
- s^0 is the initial state
- δ is the transition function (where $\delta(s, a) = s'$ is represented by: $s \rightarrow_a s'$)
- F is the set of final states.

Roughly speaking, a service is modeled as a state machine able to execute, according to the state it is in, actions taken from a shared alphabet A.

Available services These are the services that correspond to existing programs, and are the only services directly *available* to the client. We remark that available services cannot be modified: they are defined once for all and evolve according to their behavior. The only way their evolution can be driven is by executing proper legal sequences of actions. In general, we deal with many (e.g., a *community*, see below) available services $\mathcal{S}_i$ $(i = 1, \ldots, n)$, each of them, of course, modeled by a transition system $TS_i = \langle A_i, S_i, s_i^0, \delta, F_i\rangle$.

Community A finite set of available services $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ forms a *community*. The available services of a community share the same set of actions A –which is, possibly, the result of joining the action alphabets of all available services. Note that available services might be not able to perform all actions in A.

For convenience, we associate also to a community a TS, which formalizes the global behavior resulting from combining in all possible ways the behaviors of the available services in the community. Formally, the *community transition system* $TS_\mathcal{C}$ of a community $\mathcal{C}$ is the *asynchronous product* of its available services. More in details, let $TS_1, \ldots, TS_n$ be the TSs associated to the available services of $\mathcal{C}$, where $TS_i = \langle A, S_i, s_i^0, \delta_i, F_i\rangle$ $(i = 1, \ldots, n\rangle$, the *community transition system* $TS_\mathcal{C} = \langle A, S_\mathcal{C}, s_\mathcal{C}^0, \delta_\mathcal{C}, F_\mathcal{C}\rangle$ is defined as follows:

- $S_\mathcal{C} = S_1 \times \ldots \times S_n$
- $s_\mathcal{C}^0 = \langle s_1^0, \ldots, s_n^0\rangle$
- $F_\mathcal{C} = F_1 \times \ldots \times F_n$
- $\delta_\mathcal{C} \subseteq S_\mathcal{C} \times A \times S_\mathcal{C}$, where $(s_1 \times \ldots \times s_n) \rightarrow_a (s_1' \times \ldots \times s_n')$ iff:

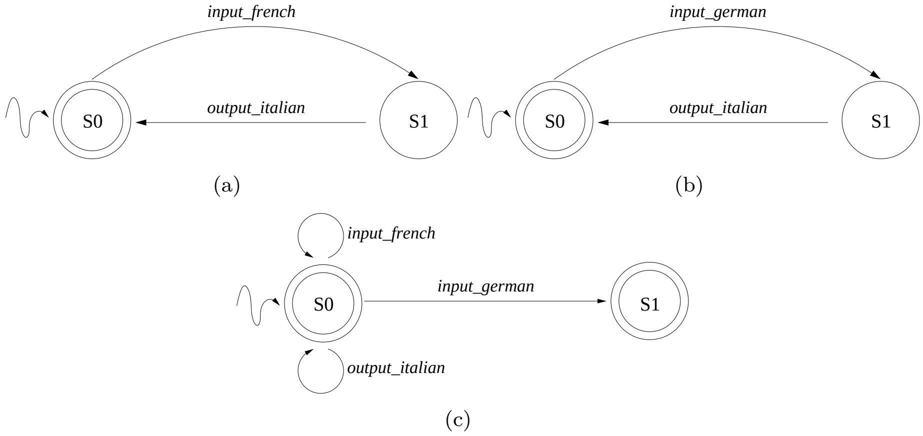

Figure 1: Example 1: Available services for a multi-lingual community

- $\exists i$ s.t. $s_i \rightarrow_a s'_i$
- $\forall j \neq i\ s'_j = s_j$

In general, despite the determinism of available services, $TS_\mathcal{C}$ may be non deterministic. Moreover, note that $TS_\mathcal{C}$ can execute a transition if and only if there exists one service among $TS_1, \ldots, TS_n$ that can do it and, hence, moves to next state according to the transition performed by such service.

Target service Our goal is to synthesize, starting from a given community, a new service that realizes a desired behavior. Such a service is called *target service* and, again, is represented by a transition system $TS_t = \langle A_t, S_t, s_t^0, \delta_t, F_t \rangle$.

Notably, the target service is not one of the available services of the community, in general. Hence, the target service has to be realized by exploiting fragments of the behaviors (computations) of the available services, since these are the only services that correspond to existing programs in the system.

The following Example makes actual the notions just introduced.

Example 1 [A multi-lingual community] Consider the services community depicted in Fig. 1, where available services provide several translation functionalities. In details, available services 1(a) and 1(b) provide, respectively, French-to-Italian and German-to-Italian translation services. For instance, think of them as web services providing a page where the user first can fill a form with some text and then can ask for its Italian translation. According to their TSs, translations can be asked for only after the form is filled out.

Similarly, the available service 1(c) provides French-to-Italian translation functionalities, besides allowing for some further operations –such as, e.g., finding synonyms– when German text is introduced (indeed, such operations are compacted into a single state, `S1`, since not relevant for our purposes). Differently from previous services, Italian translation can be performed even if no text is explicitly introduced,

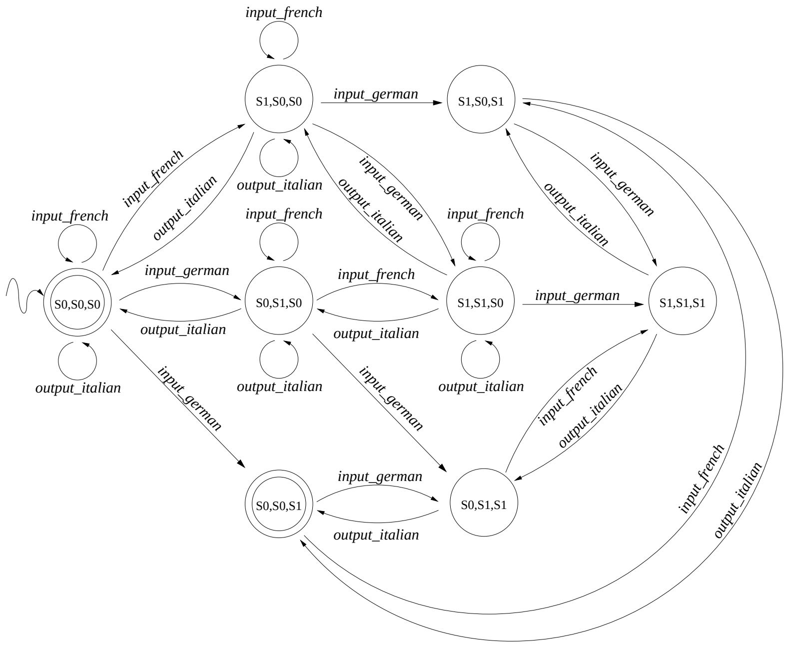

Figure 2: Example 1: community transition system

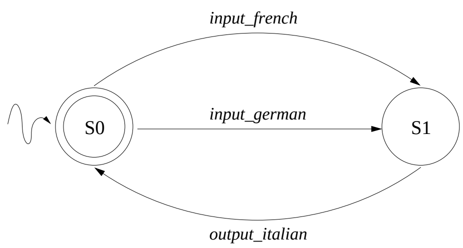

Figure 3: Example 1: target service

as shown by the looping edge on state `S0`, labeled by action *output_italian* –we can imagine, e.g., that a buffer, initially filled out with some default text, is used to record the last translated input.

We will refer to TSs associated to services 1(a), 1(b) and 1(c) by means of subscripts a, b and c, respectively. For instance, TS associated to service 1(a) is referred to as $TS_a = \langle A_a, S_a, s_a^0, \delta_a, F_a \rangle$. The community TS is represented by $TS_\mathcal{C} = \langle A_\mathcal{C}, S_\mathcal{C}, s_\mathcal{C}^0, \delta_\mathcal{C}, F_\mathcal{C} \rangle$

Finally, in Fig. 2 the *community transition system* is shown which describes the behavior of the community seen as a whole system, where actions are performed by exactly one available service at a time. State labels are triples $\langle s_a, s_b, s_c \rangle \in S_a \times S_b \times S_c$ representing the state of each service after actions execution. Note that the community TS is non-deterministic.

Given such community, we are interested in synthesizing or, better said, *composing*, the *target* service depicted in Fig. 3, which allows for translating either French or German input text to Italian. □

3. Service Composition

Intuitively, the service composition problem can be stated as follows:

> Given a target service and a community, synthesize a *composition*, i.e., a suitable function that delegates actions, requested by the client to the target service, to the available services in the community (which are the only services actually corresponding to existing programs).

As already discussed, both available and target services are represented by transition systems over a common actions alphabet A. Recall that *(i)* before any interaction takes place, each available service is in its initial state and *(ii)* a service can be left only if it is in a final state. Basically, composing a target service amounts to mimicking the desired (target) behavior by properly instructing, for each action chosen by the client (coherently with the target service) a particular available service to perform the requested action. Of course, each time a service is to be selected for executing some action, the choice is constrained by the current state, as the result of the actions done so far, of each available service (recall that a service evolves each time it interacts with some client). In addition, it obviously depends also from future actions that coherently with the target behavior, can be later requested by a client.

In order to make such intuition precise we first introduce the notion of *execution tree*.

Execution trees TSs provide a compact description of service abilities, but take into account no issue concerning actual evolution. If, on one hand, TSs describe *which* actions a service can execute and how its state changes, on the other hand, they do not keep track of *how* states are reached. As a matter of fact, in general, a given state may result from the execution of different action sequences, and the

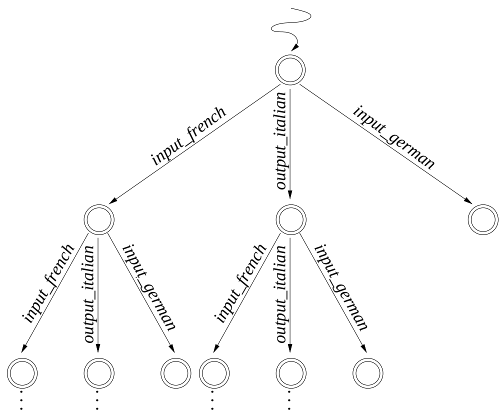

Figure 4: Example 1: execution tree of service c)

state itself holds no information about which of them has been actually executed. Since this aspect is crucial for our purposes, a formal definition is required.

The actual evolution of a service can be described by an *execution tree*. Intuitively, it is a structure obtained by "unfolding" the TS associated to the service itself. More formally, given a service $\mathcal{S}$ and its respective transition system $TS = \langle A, S, s^0, \delta, F\rangle$, an execution tree for $\mathcal{S}$ is a pair $\langle \mathcal{T}, final\rangle$, where $\mathcal{T}$ is a tree over A (i.e., a prefixed closed set of string over A) and $final$ is a boolean function over nodes of $\mathcal{T}$. Both $\mathcal{T}$ and $final$ are inductively defined by making use of an auxiliary function $m_{TS} : \mathcal{T} \to S$, as follows:

- $\varepsilon \in \mathcal{T}$, and $m_{TS}(\varepsilon) = s_0$, i.e., m_{TS} associates the root ε of $\mathcal{T}$ to the initial state s_0 of TS;
- let $x \in \mathcal{T}$, and $m_{TS}(x) = s$ where $s \in S$: if $s \to_a s'$ then $x{\cdot}a \in \mathcal{T}$, i.e., x has an a successor, and $m_{TS}(x{\cdot}a) = s'$;
- $final(x) = \texttt{true}$ iff $m_{TS}(x) \in F$.

Observe that each node of $\mathcal{T}$ is a sequence of actions $x = a_1 \cdot \cdots \cdot a_k$ allowed in TS, starting from the initial state. Each of such sequences is called *history* for TS. In other words, each node $x = a_1 \cdot \cdots \cdot a_k$ of execution tree $\mathcal{T}$ represents a history for TS. Given a history $a_1 \cdot \cdots \cdot a_k$, we do know the state of TS after its execution, starting from the initial state, namely $m_{TS}(a_1 \cdot \cdots \cdot a_k)$. Also, notice that, given a node $x = a_1 \cdot \cdots \cdot a_k$ of $\mathcal{T}$, the successor nodes of x, namely $x{\cdot}a^1_{k+1}, \ldots, x{\cdot}a^\ell_{k+1}$, tell us which actions, namely $a^1_{k+1}, \ldots, a^\ell_{k+1}$, are allowed in the current state of TS, that is, the state reached from the initial state through history $a_1 \cdot \cdots \cdot a_k$. Furthermore, function $final(.)$ tells us whether through the history x, TS has reached a final state, i.e., whether $final(x) = \texttt{true}$.

Example 2 Fig. 4 depicts the execution tree generated by system 1(c). Note that, coherently with its respective transition system: *(i)* every state is final, *(ii)* action *input_german* always leads to a sink node where no further action can be performed and *(iii)* whenever execution is in a state where either *input_french* or *output_italian* can be performed, any arbitrary sequence of such actions is allowed. Construction of execution trees for systems 1(a) and 1(b) is straightforward. □

Composition With the notion of execution tree of a TS in place, we can formally define service composition. The crux notion is that of *composition labeling* that formalizes the idea of assigning actions to services.

Definition 1 (Composition labeling) *Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community of available services, $\mathcal{S}_t$ be the target service and $\mathcal{T}_i^{\mathcal{S}} = \langle \mathcal{T}_i, final_i \rangle$ be the execution tree for $\mathcal{S}_i$ ($i = 1, \ldots, n, t$). A* composition labeling *of $\mathcal{T}_t^{\mathcal{S}}$ wrt $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$ is a function* $\textsc{clab} : \mathcal{T}_t^{\mathcal{S}} \rightarrow \mathcal{T}_1^{\mathcal{S}} \times \cdots \times \mathcal{T}_n^{\mathcal{S}}$ *that satisfies the following conditions:*

1. $\textsc{clab}(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$;
2. *for every node $x \in \mathcal{T}_t$, let $\textsc{clab}(x) = \langle x_1, \ldots, x_n \rangle$; then, for all $a \in A$ such that $x{\cdot}a \in \mathcal{T}_t$, $\textsc{clab}(x{\cdot}a) = \langle y_1, \ldots, y_n \rangle$, where $y_i = x_i{\cdot}a$ for exactly one $i \in [1, \ldots, n]$ (if service $\mathcal{S}_i$ performs interaction a) and $y_j = x_j$ otherwise.*
3. *for every node $x \in \mathcal{T}_t$, if $final_t(x) = \texttt{true}$ and $\textsc{clab}(x) = \langle x_1, \ldots, x_n \rangle$, then $final_i(x_i) = \texttt{true}$ for $i = 1, \ldots, n$.*

Intuitively, CLAB labels each node of the target service execution tree $\mathcal{T}_t^{\mathcal{S}}$ with a tuple $\langle x_1, \ldots, x_n \rangle$, where the generic component x_i ($i = 1, \ldots, n$) denotes the current node of the execution tree $\mathcal{T}_i^{\mathcal{S}}$, i.e., the history of actions executed so far, starting from the initial state, by i-th available service. Requirement (1) states that all services start from the beginning of their computation, i.e., their initial state; requirement (2) constrains each action of the target service to be executed by exactly one available service (in its current state, which results from its history so far), while the other services remain still; and finally, requirement (3) allows for leaving the target service only if all available services are in a final configuration. Summing up, CLAB relates, in a step-by-step fashion, the evolution of the target service to the evolution of available services, by suitably delegating in a step-by-step fashion actions requested to the target services to one of the available service.

Given a composition labeling CLAB, one can orchestrate the n available services to mimic the target service $\mathcal{S}_t$ by stepping each available service according to what specified by CLAB itself. Thus, *service composition* can be formally defined as follows:

Definition 2 (Service composition) *A composition of the services in the community $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ realizing the target service $\mathcal{S}_t$ is a function* $\textsc{comp} : \mathcal{T}_t^{\mathcal{S}} \rightarrow \{1, \ldots, n\} \cup \bot$ *such that*

- $\textsc{comp}(\varepsilon) = \bot$
- $\textsc{comp}(x{\cdot}a) = i$, *where* $\textsc{clab}(x) = \langle y_1, \ldots, y_i, \ldots, y_n \rangle$ *and* $\textsc{clab}(x{\cdot}a) = \langle y_1, \ldots, y_i{\cdot}a, \ldots, y_n \rangle$, *i.e.,* $\textsc{clab}(x)$ *and* $\textsc{clab}(x{\cdot}a)$ *are identical except for the i-th component that from y_i in* $\textsc{clab}(x)$ *becomes $y_i{\cdot}a$ in* $\textsc{clab}(x{\cdot}a)$.

Observe that, by definition, given a composition labeling CLAB we get the corresponding composition COMP. The vice-versa is also true, given a composition COMP, it is immediate to get the corresponding composition labeling CLAB as follows:

- $\text{CLAB}(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$;
- for every node $x \in \mathcal{T}_t$, let $\text{CLAB}(x) = \langle x_1, \ldots, x_n \rangle$; then for all $a \in A$ such that $x{\cdot}a \in \mathcal{T}_t$ $\text{CLAB}(x{\cdot}a) = \langle y_1, \ldots, y_n \rangle$, where $y_i = x_i{\cdot}a$ if $\text{COMP}(x{\cdot}a) = i$, and $y_j = x_j$ otherwise.

Computational complexity characterization Composition, as defined above, has already been studied in [4, 5]. In particular,the computational complexity characterization of the problem is known. The upper bound was establish in [4]:
Theorem 1 ([4]) *Checking the existence of a composition of the services in a community $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ that realizes a target service $\mathcal{S}_t$ can be done in EXPTIME.*
A matching lower bound was recently proved by Muscholl and Walukiewicz:
Theorem 2 ([23]) *Checking the existence of a composition of the services in a community $\mathcal{C} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle$ that realizes a target service $\mathcal{S}_t$ is EXPTIME-hard.*
In other words the checking the existence of a composition is an EXPTIME-complete problem.

Notably, in [4, 5] an actual synthesis technique for computing the composition is presented. Such a technique is based on a polynomial reduction to satisfiability in Propositional Dynamic Logic [15]. Here, however, we do not relay on such a technique. Instead, we develop a new composition synthesis technique based on the notion of simulation.

4. Composition and Simulation

Now, we illustrate the basic result of this paper: we show that checking for existence of a services composition can be done by checking for the existence of a simulation relation between the target and the community TSs. We start by defining the notion of simulation relation [22] in our context.

Definition 3 (Simulation relation) *Given two transition systems TS_t and $TS_{\mathcal{C}}$, a simulation relation of TS_t by $TS_{\mathcal{C}}$ is a relation $R \subseteq S_t \times S_{\mathcal{C}}$, such that:*

$R(s_t, s_{\mathcal{C}})$ implies:

1. *if $s_t \in F_t$ then $s_{\mathcal{C}} \in F_{\mathcal{C}}$;*
2. *for all transitions $s_t \rightarrow_a s'_t$ in TS_t there exists a transition $s_{\mathcal{C}} \rightarrow_a s'_{\mathcal{C}}$ in $TS_{\mathcal{C}}$ and $R(s'_t, s'_{\mathcal{C}})$.*

The definition says that state s_t of TS_t is in a simulation relation R with $s_{\mathcal{C}}$ of $TS_{\mathcal{C}}$ if: *(i)* if s_t is final then also $s_{\mathcal{C}}$ is final; *(ii)* for every action a and state s'_t, if s_t can make a transition to s'_t with action a, then also $s_{\mathcal{C}}$ can make a transition to some $s'_{\mathcal{C}}$ with action a, in such a way that s'_t is still in the same simulation relation R with $s'_{\mathcal{C}}$. Observe the *coinductive* nature of such a definition: indeed the definition is cyclic but with no base case.

Definition 4 *Let TS_t be the transition system representing the target service, and $TS_\mathcal{C}$ be the community transition system. A state $s_t \in S_t$* is simulated by *a states $s_\mathcal{C} \in S_\mathcal{C}$ (or $s_\mathcal{C}$* simulates *s_t), denoted $s_t \preceq s_\mathcal{C}$, iff there exists a simulation R of TS_t by $TS_\mathcal{C}$ s.t $R(s_t, s_\mathcal{C})$.*

Observe that the relation $\preceq$ is itself a simulation relation and in fact $\preceq$ is the *largest simulation relation*, indeed by the definition above all simulation relations are contained in $\preceq$.

Definition 5 *TS_t* is simulated by *$TS_\mathcal{C}$ (or $TS_\mathcal{C}$* simulates *TS_t) iff $s_t^0 \preceq s_\mathcal{C}^0$.*

Example 3 [Example 1, continued] Consider the target (Fig. 3) and the community (Fig. 2) services of Example 1. In Figure 5 a simulation of the latter service by the former one is given, where dashed lines associate each state of the target TS to those of the community TS it is simulated by. Therefore, e.g., state ⟨`S1,S0,S0`⟩ of $TS_\mathcal{C}$ *simulates* state `S1` of TS_t as well as state `S0` is simulated by both ⟨`S0,S0,S0`⟩ and ⟨`S0,S0,S1`⟩. Note that, in general, there may exist several simulations. The one shown in Figure 5 represents, in facts, the *largest* one, i.e., the relation $\preceq$. □

Theorem 3 below, shows how checking the existence of a service composition can be reduced to checking that the target transition system is simulated by the community transition system. To prove it we introduce two lemmas.

Lemma 1 *Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community, $\mathcal{S}_t$ a target service, and* CLAB *a composition labeling of $\mathcal{T}_t^\mathcal{S}$ wrt $\mathcal{T}_1^\mathcal{S}, \ldots, \mathcal{T}_n^\mathcal{S}$. Then the relation $R \subseteq S_t \times S_\mathcal{C}$ defined as $R = \{\langle s_t, s_c\rangle \mid \exists x, x_1, \ldots, x_n : m_{TS_t}(x) = s_t, \textsc{clab}(x) = \langle x_1, \ldots, x_n\rangle, \langle m_{TS_1}(x_1), \ldots, m_{TS_n}(x_n)\rangle = s_c\}$ is a simulation relation of TS_t by $TS_\mathcal{C}$ such that $R(s_t^0, s_\mathcal{C}^0)$.*

Proof. The following arguments prove that R is a simulation:

- Since $\textsc{clab}(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon\rangle$ by definition of CLAB, then $R(s_t^0, s_\mathcal{C}^0)$ holds by definition of R.
- Consider a final node $x \in \mathcal{T}_t$ associated to a final state $s_t \in F_t \subseteq S_t$ by $m_{TS_t}(x) = s_t$. Recall that s_t is final iff x does. By definition of R, x is associated, by CLAB, to a tuple $\langle x_1, \ldots, x_n\rangle$ such that $\langle m_{TS_1}(x_1), \ldots, m_{TS_n}(x_n)\rangle = s_\mathcal{C}$. By definition of CLAB, being x final, also $x_1, \ldots, x_n$ do. By definition of m_{TS_i} each s_i is final, therefore $s_\mathcal{C}$ is final.
- Let $m_{TS_t}(x) = s_t$, $x' = x{\cdot}a$, $s_t \rightarrow_a s_t'$ and $m_{TS_t}(x') = s_t'$. By definition of R, $\textsc{clab}(x) = \langle x_1, \ldots, x_n\rangle$ and $\langle m_{TS_1}(x_1), \ldots, m_{TS_n}(x_n)\rangle = s_\mathcal{C}$. By definition of CLAB, $\textsc{clab}(x{\cdot}a) = \langle x_1', \ldots, x_n'\rangle$, where for one $i \in [1, \ldots, n]$, we get $x_i' = x_i{\cdot}a$, and for all other $j \in [1, \ldots, n]$ with $j \neq i$, we get $x_j' = x_j$. Finally, by definition of m_{TS_i}, $m_{TS_i}(x_i') = s_i'$ iff $s_i \rightarrow_a s_i'$. Hence, $s_\mathcal{C} \rightarrow_a s_\mathcal{C}'$ and, consequently, $R(s_t', s_\mathcal{C}')$ holds. □

The lemma above *constructively* states that, given a target service $\mathcal{S}_t$ and a community $\mathcal{C}$, for every composition labeling of the execution tree associated to $\mathcal{S}_t$ by the execution trees of community services, it is always possible to build a relation R which is a simulation of TS_t by $TS_\mathcal{C}$.

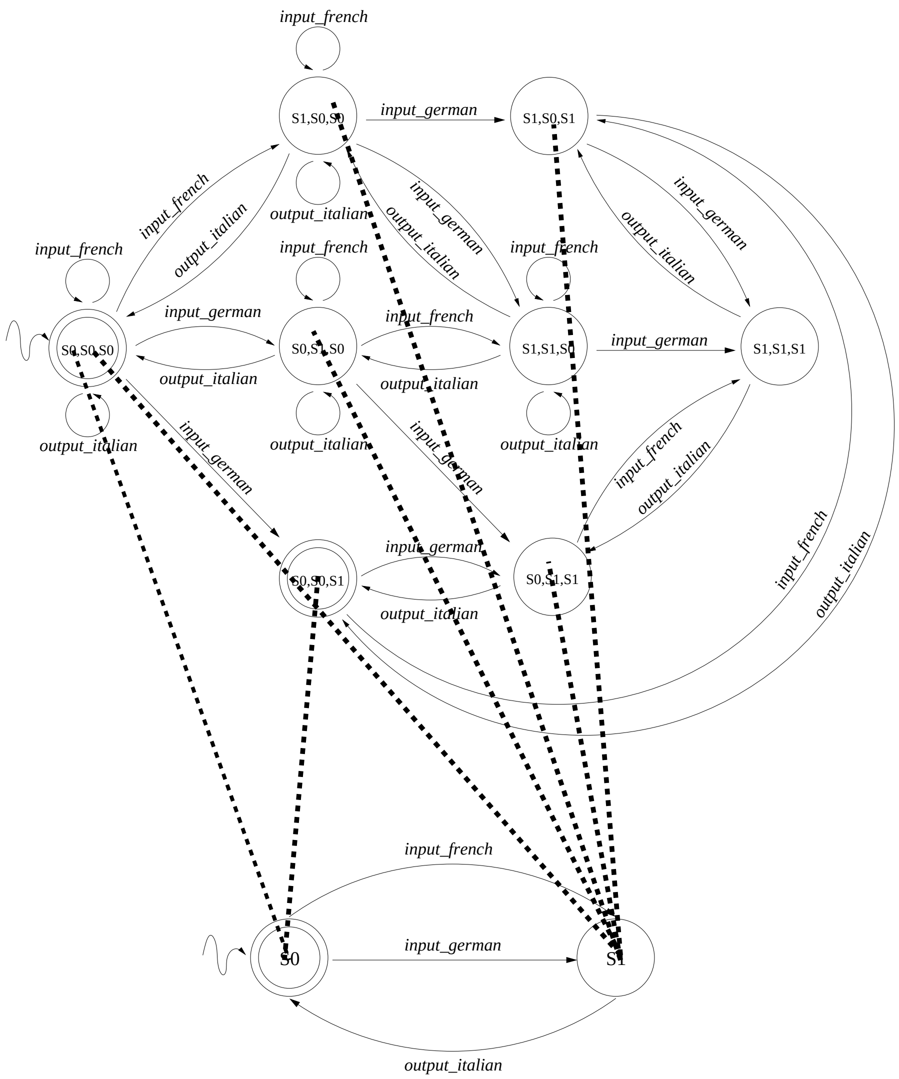

Figure 5: Example 3: A simulation relation for Example 1

Lemma 2 *Let $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be a community, $\mathcal{S}_t$ a target service, and R a simulation relation of TS_t by $TS_\mathcal{C}$ such that $R(s_t^0, s_\mathcal{C}^0)$. Then, there exists a composition labeling* CLAB *a composition labeling of $\mathcal{T}_t^\mathcal{S}$ wrt $\mathcal{T}_1^\mathcal{S}, \ldots, \mathcal{T}_n^\mathcal{S}$.*

Proof. Let R be a simulation of TS_t by $TS_\mathcal{C}$ such that $R(s_t^0, s_\mathcal{C}^0)$ where $s_\mathcal{C}^0 = \langle s_1^0, \ldots, s_n^0 \rangle$. From R we can build a labeling function CLAB : $\mathcal{T}_t^\mathcal{S} \to \mathcal{T}_1^\mathcal{S} \times \cdots \times \mathcal{T}_n^\mathcal{S}$, by induction on the level of nodes in $\mathcal{T}_t$, which shows that a composition does exist. Recall that *(i)* R associates each state of TS_t to a tuple of states from $TS_1 \times \ldots \times TS_n$ (that is, the set of community TS states) and *(ii)* a mapping m_{TS_i} associates each node of $\mathcal{T}_i$ to a corresponding state of TS_i ($i = 1, \ldots, n, t$). We proceed as follows:

- Base case.
  $m_{TS_t}(\varepsilon) = s_t^0$, i.e. the root of $\mathcal{T}_t$ is labeled with the initial state of TS_t, and analogously for each TS_i. Since R is a simulation, we have that $R(s_t^0, \langle s_1^0, \ldots, s_n^0 \rangle)$. Therefore, we define CLAB$(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$.

- Inductive hypothesis:
  Let $m_{TS_t}(x) = s_t$ and let $R(s_t, \langle s_1, \ldots, s_n \rangle)$. Let CLAB$(x) = \langle x_1, \ldots, x_n \rangle$, where $m_{TS_i}(x_i) = s_i$.

- Induction step.
  Let $x' = x \cdot a$ be a successor node of x. If such a node exists, there exists also a transition $s_t \to_a s'_t$ such that $m_{TS_t}(x') = s'_t$. Therefore, since $R(s_t, \langle s_1, \ldots, s_n \rangle)$ holds by inductive hypothesis, then a tuple[a] $\langle s'_1, \ldots, s'_n \rangle$ exists such that $R(s'_t, \langle s'_1, \ldots, s'_n \rangle)$. Such a tuple, by definition of $TS_\mathcal{C}$, must be such that for one $i \in [1, \ldots, n]$, we have $s_i \to_a s'_i$ and for all other $j \in [1, \ldots, n]$ with $j \neq i$, we have that $s'_j = s_j$. Hence, we can define CLAB$(x') = \langle x'_1, \ldots, x'_n \rangle$, where $m_{TS_t}(x'_i) = s'_i$ and:

  - if $s'_i = s_i$ then $x'_i = x_i$
  - if $s_i \to_a s'_i$ then $x'_i = x_i \cdot a$

Finally, recall that each $\mathcal{T}_t$'s final node is associated, through m_{TS_t}, to exactly one TS_t's final state. Let x be one of such nodes and let $m_{TS_t}(x) = s_t$. Since R is a simulation, it relates s_t to some tuple(s) $\langle s_1, \ldots, s_n \rangle$, where each component is a final state for the TS it refers to. Hence being CLAB$(x) = \langle x_1, \ldots, x_n \rangle$, with $m_{TS_i}(x_i) = s_i$, we get that, $x_1, \ldots, x_n$ are final. Concluding we get that, CLAB defined as above is indeed a composition labeling. □

This lemma says that given a simulation R of TS_t by $TS_\mathcal{C}$, the whole set of composition labelings which realize the target service can be always defined. Note that such set is not a singleton since, in general, R associates a TS_t's state to many (possibly one) $TS_\mathcal{C}$'s states. Observe that also the proof of this lemma is constructive.

As a direct consequence of Lemmas 1 and 2, we get our theorem.

[a] In general, R associates several n-tuples to s'_t since $TS_\mathcal{C}$ may be non deterministic.

Theorem 3 *A composition of the services in the community* $\mathcal{C} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ *realizes the target service* $\mathcal{S}_t$ *if and only if* TS_t *is simulated by* $TS_{\mathcal{C}}$*.*

Proof. By definition of composition, it suffices to prove that there exists a composition labeling of $\mathcal{T}_t^{\mathcal{S}}$ wrt $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$ if and only if TS_t is simulated by $TS_{\mathcal{C}}$, which is a consequence of Lemma 1 for "$\Rightarrow$" direction and Lemma 2 for "$\Leftarrow$" direction. $\Box$

Theorem 3 gives us an straightforward method to check for the existence of composition, namely:

- compute the maximal simulation relation $\preceq$ of TS_t by $TS_{\mathcal{C}}$;
- check that $\langle s_t^0, s_{\mathcal{C}}^0 \rangle$ is in such a relation.

Observe that such a method is quite different from the one in [4] which was based on a polynomial reduction to satisfiability in Propositional Dynamic Logic [15].

From the computational point of view, we recall that checking the existence of a simulation relation between two (states of two) transition systems can be done in polynomial time in the size of the transition systems –moreover well developed techniques exists for computing simulation, such as those in [16, 27, 10]. Since in our case the number of states of $TS_{\mathcal{C}}$ is exponential in the size (i.e., the number of states) of $TS_1, \ldots, TS_n$, we get that we can check for the existence of a composition using simulation in exponential time. Considering that the problem is EXPTIME-complete, we get that indeed checking existence of compositions via simulation is indeed *optimal* wrt worst-case complexity.

5. Synthesizing Composition via Simulation

Theorem 3 closely relates the notion of *simulation relation* to the one of *service composition* showing, ultimately, that finding a service composition corresponds to finding a simulation relation between two particular –the target and the community– transition systems and vice-versa. However, no procedure is given for *actually* synthesizing an *orchestrator* that implements such a composition by properly assigning action executions to available services.

In this section, we show that if a service composition exists, it can be used to synthesize an *orchestrator*. To this end, we refer to an abstract structure called *orchestrator generator*, or simply *OG*. Intuitively, the *OG* is a program that returns, for each state reached by the community in realizing a target history, the set of available services capable of performing the (target-conformant) action the client next requests. As shown below, *OG* is directly obtained from the maximal simulation relation between the target and the community TSs.

Definition 6 (Orchestrator Generator, *OG*) *Let* $\mathcal{S}_t$ *be a target service and* $\mathcal{C} = \{\mathcal{S}_1, \ldots, S_n\}$ *be a community of available services such that* TS_t *is simulated by* $TS_{\mathcal{C}}$*. The* orchestrator generator *(*OG*) for* TS_t *and* $TS_{\mathcal{C}}$ *is a tuple* $OG = \langle A, [1, \ldots, n], S_r, s_r^0, \omega_r, \delta_r, F_r \rangle$*, where:*

1. *A is the finite set of community actions;*

2. $[1,\ldots,n]$ *is the set of available services indices;*

3. $S_r = S_t \times S_1 \times \ldots \times S_n$ *is the set of OG states;*

4. $s_r^0 = \langle s_t^0, s_1^0, \ldots, s_n^0 \rangle$ *is the OG's initial state;*

5. $F_r = \{(s_t, s_1, \ldots, s_n) \mid s_t \in F_t\}$ *is the set of OG's final states;*

6. $\omega_r : S_r \times A \mapsto 2^{[1,\ldots,n]}$ *is the* service selection function*:*
   let $s_r = \langle s_t, s_1, \ldots, s_n \rangle \in S_r$*,* $\omega_r(s_r, a)$ *is defined iff*

   - $s_t \preceq \langle s_1, \ldots, s_n \rangle$ *and*
   - *there exists* s_t' *s.t* $s_t \to_a s_t'$*;*

   in such case, $\omega_r(s_r, a) = \{k \mid \exists\ s_k'.\ s_k \to_a s_k' \wedge s_t' \preceq \langle s_1, \ldots, s_k', \ldots, s_n \rangle\}$*;*

7. $\delta_r : S_r \times A \times [1,\ldots,n] \to S_r$ *is the* transition function*.*
   $\delta_r(s_r, a, k)$ *is defined iff* $k \in \omega_r(s_r, a)$*, as follows:*

   $\delta_r(s_r, a, k) = s_r'$*, where* $s_r' = \langle s_t', s_1, \ldots, s_k', \ldots, s_n \rangle$*,* $s_t \to_a s_t'$ *and,* $s_k \to_a s_k'$*.*

Intuitively, OG is a finite state machine that, at each point, given a (target-conformant) action a, outputs (function ω_r) the set of services which can perform a next according to the maximal simulation relation $\preceq$. For each choice of one of such services it progresses to the next state (function δ_r).

Once we have OG, we get *orchestrators* by choosing, at each point, one of the outputs of ω_r. Formally, we define a *(generated) orchestrator* as follows:

Definition 7 (Generated Orchestrator) *Given an orchestrator generator OG for* TS_t *and* $TS_{\mathcal{C}}$*, defined as above, a* generated orchestrator *is a function* ORCH : $\mathcal{T}_t \to [1,\ldots,n] \cup \bot$*, inductively defined as follows:*

- $\text{ORCH}(\varepsilon) = \bot$*;*
- *if* $x{\cdot}a \in \mathcal{T}_t$*, then* $\text{ORCH}(x{\cdot}a) = i \in \omega_r(\sigma_r^{\text{ORCH}}(x), a)$*, where:*
  $\sigma_r^{\text{ORCH}} : \mathcal{T}_t \to S_r$ *is the mapping function between nodes of* $\mathcal{T}_t$ *and corresponding states of* S_r*, defined as follows:*
  - $\sigma_r^{\text{ORCH}}(\varepsilon) = s_r^0$*;*
  - *if* $x{\cdot}a \in \mathcal{T}_t$ *then* $\sigma_r^{\text{ORCH}}(x{\cdot}a) = \delta_r(\sigma_r^{\text{ORCH}}(x), a, \text{ORCH}(x{\cdot}a))$

A generated orchestrator is, basically, a function which selects, at each point, an available service for executing the action requested by a (target-conformant) client at that point. In order to guarantee the selected service to be actually capable of executing the assigned action, orchestrator assignments must belong to the set defined by ω_r, at each step. Note that such set depends on σ_r^{ORCH} which, in turns, depends on ORCH itself. σ_r^{ORCH} maps each node of target service execution tree, that is a target history, into the state reached by the community TS when such history is actually executed. Since, also, ORCH depends on σ_r^{ORCH}, both functions are defined by mutual induction, through ω_r. Note how induction is *well-founded*, since i-th step value of ORCH depends, through ω_r on $(i-1)$-th step value of σ_r^{ORCH}.

A generated orchestrator defines a labeling of $\mathcal{T}_t$ by tuples of nodes from $\mathcal{T}_1 \times \ldots \times \mathcal{T}_n$, representing the correspondence between a particular history of the target behavior and those of community services. Such correspondence, of course, is strictly related to the history of service assignments, that is, ultimately, to ORCH. Formally, an *orchestrator labeling* is defined as follows:

Definition 8 (Orchestrator Labeling) *Given an orchestrator generator OG for TS_t and $TS_{\mathcal{C}}$ and a respective generated orchestrator* ORCH, *defined as above, an* orchestrator labeling OLAB *of $\mathcal{T}_t^{\mathcal{S}}$ by $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$ is defined wrt* ORCH *as a function* OLAB $: \mathcal{T}_t \rightarrow \mathcal{T}_1 \times \ldots \mathcal{T}_n$ *which satisfies the following conditions:*

- OLAB$(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$;
- *for every node $x \in \mathcal{T}_t$, let* OLAB$(x) = \langle x_1, \ldots, x_n \rangle$; *then for all $a \in A$ such that $x \cdot a \in \mathcal{T}_t$,* OLAB$(x \cdot a) = \langle y_1, \ldots, y_n \rangle$ *where $y_i = x_i \cdot a$ if* ORCH$(x \cdot a) = i$ *and $y_j = x_j$ otherwise.*

Example 4 [Example 3, continued] As an instance of Orchestrator Generator (OG), consider Figure 6, where a graphical representation of the $OG = \langle A, [1, \ldots, n], S_r, s_r^0, \omega_r, \delta_r, F_r \rangle$ obtained by the simulation relation of Figure 5 is shown. According to Definition 6, nodes are labeled by four components representing, respectively, states of target, a), b) and c) services. Note that it includes two disconnected components. Obviously, only the one containing the initial state is relevant, as the other one(s) cannot be reached, all services being initially assumed in their initial state. Edges are labeled by pairs I/O, where $I \in A \times [1, \ldots, n]$ and $O \in 2^{[1, \ldots, n]}$, with the following semantics: an edge e connects node s to node s' with label $\langle a, i \rangle / O$ iff $\omega_r(s, a) = O$, $i \in O$ and $\delta_r(s, a, i) = s'$. Starting from this OG, several orchestrators can be obtained, depending on the service selected at each step for performing each interaction. Intuitively, generating an orchestrator corresponds to unfolding an OG and labeling the resulting edges by choosing one among the services proposed by the selection function ω_r. For instance, in Figure 7(a) two different orchestrators are reported. Edges are labeled with pairs a/i, where $a \in A$ and $i \in [1, \ldots, n]$ represent, respectively, the client requested action and the *orch*'s available service choice. ☐

Now, we show that all generated orchestrators lead to a composition of available services that realizes the client request. Even more importantly, the vice-versa holds: every composition can be obtained by suitably choosing, at each step, one element from those in OG's selection function ω_r. In other words, the maximal simulation virtually contains all compositions. orchestration and composition labelings.

Theorem 4 *If* OLAB *is an orchestrator labeling of $\mathcal{T}_t^{\mathcal{S}}$ by $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$, then* OLAB *is a composition labeling of $\mathcal{T}_t^{\mathcal{S}}$ by $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$.*

Proof. We need to show that any orchestrator labeling OLAB fulfills requirements of Definition 1. Let OLAB be an orchestrator labeling of $\mathcal{T}_t^{\mathcal{S}}$ by $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$.

1. By definition of orchestrator labeling, OLAB$(\varepsilon) = \langle \varepsilon, \ldots, \varepsilon \rangle$;

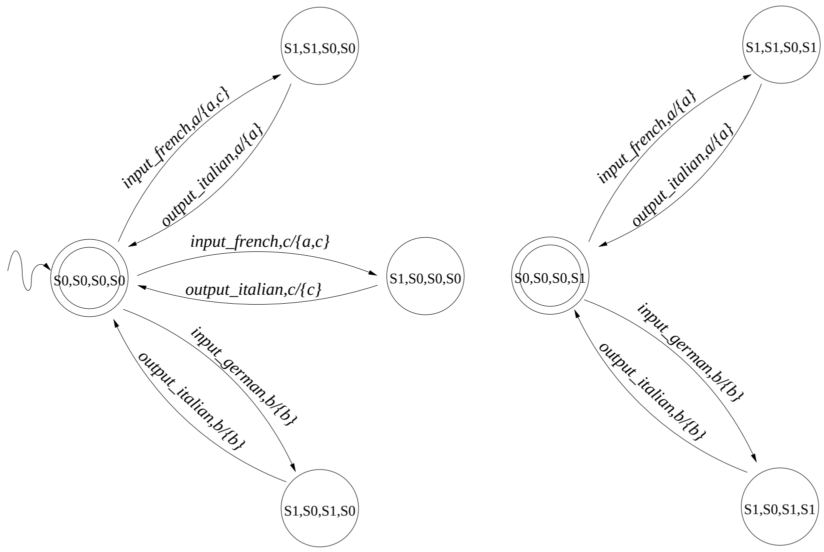

Figure 6: OG for Example 3

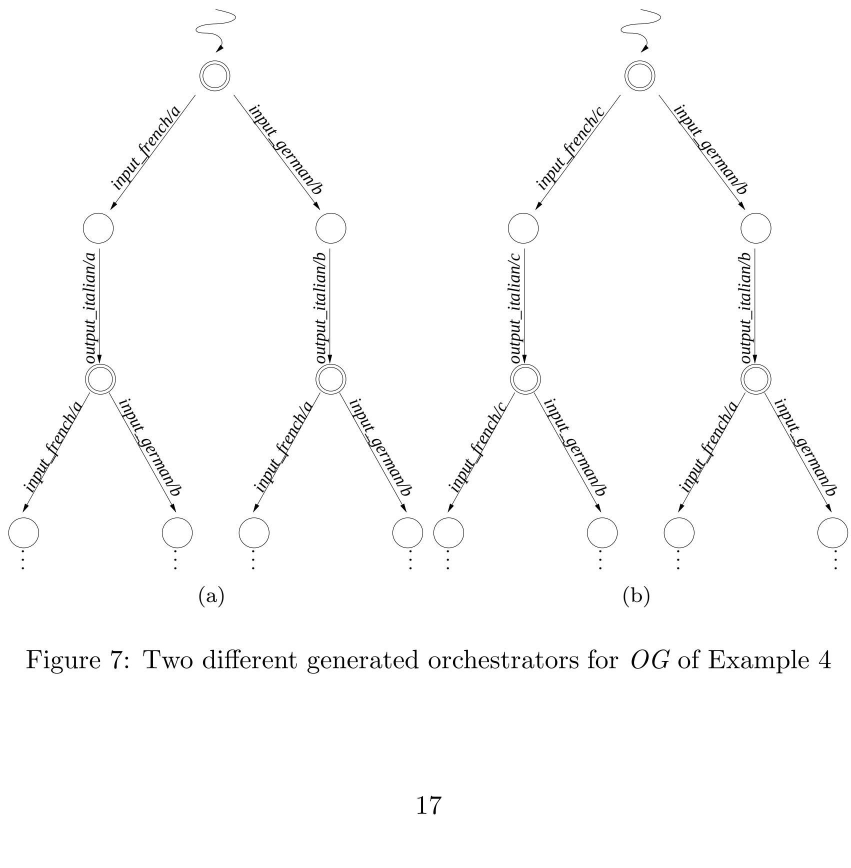

Figure 7: Two different generated orchestrators for OG of Example 4

2. By definition of orchestrator labeling, for every node $x \in \mathcal{T}_t$, if $\text{OLAB}(x) = \langle x_1, \ldots, x_n \rangle$, then for all $a \in A$ such that $x{\cdot}a \in \mathcal{T}_t$, $\text{OLAB}(x{\cdot}a) = \langle y_1, \ldots, y_n \rangle$, where $y_i = x_i{\cdot}a$ if $\text{ORCH}(x{\cdot}a) = i$ and $y_j = x_j$ otherwise. Since OLAB is defined wrt an orchestrator ORCH (Definition 8), then $\text{ORCH}(i)$ is defined and identifies the only service capable of performing action a.

   Moreover, let $\text{OLAB}(x{\cdot}a) = \langle y_1, \ldots, y_n \rangle$, if $m_{TS_t}(x{\cdot}a) = s_t$ and $m_{TS_i}(y_i) = s_i$ $(i = 1, \ldots, n)$ then, from *orch* and OLAB definitions, it follows that $s_t \preceq \langle s_1, \ldots, s_n \rangle$.

3. We need to prove that if $x \in \mathcal{T}_t$ is final and $\text{OLAB}(x) = \langle x_1, \ldots, x_n \rangle$ then all x_i $(i = 1, \ldots, n)$ are final, as well.
   By definition of m_{TS_i}, a node $x_i \in \mathcal{T}_i$ is final iff $m_{TS_i}(x_i)$ is final for TS_i $(i = 1, \ldots, n, t)$. Of course, if $x \in \mathcal{T}$ is final then s_t also does. Hence, since $s_t \preceq \langle s_1, \ldots, s_n \rangle$, where $m_{TS_t}(x) = s_t$ and $m_{TS_i}(y_i) = s_i$ $(i = 1, \ldots, n)$, s_i is final for its respective TS_i $(i = 1, \ldots, n)$ and, consequently, x_i is final for its respective execution tree $\mathcal{T}_i$. □

Theorem 5 *If* CLAB *is a composition labeling of* $\mathcal{T}_t^{\mathcal{S}}$ *by* $\mathcal{T}_1^{\mathcal{S}}, \ldots, \mathcal{T}_n^{\mathcal{S}}$*, then* CLAB *is an orchestrator labeling defined wrt an orchestrator* ORCH *generated by the OG for* TS_t *and* $TS_{\mathcal{C}}$*.*

Proof. First, observe that, due to Lemma 1, $TS_{\mathcal{C}}$ can simulate TS_t and, therefore, the orchestrator generator $OG = \langle A, [1, \ldots, n], S_r, s_r^0, \omega_r, \delta_r, F_r \rangle$ for TS_t and $TS_{\mathcal{C}}$ exists. Now, consider the function $\text{ORCH} : \mathcal{T}_t \to [1, \ldots, n] \cup \bot$, defined as follows:

1. $\text{ORCH}(\varepsilon) = \bot$;

2. $\text{ORCH}(x) = k$ iff there exists a node $x \in \mathcal{T}_t$ and an action $a \in A$ such that i) $x{\cdot}a \in \mathcal{T}_t$, ii) $\text{CLAB}(x) = \langle x_1, \ldots, x_n \rangle$ and iii) $\text{CLAB}(x{\cdot}a) = \langle x_1, \ldots, x_k{\cdot}a, \ldots, x_n \rangle$ for exactly one $k \in [1, \ldots, n]$.

Referring to Definition 7, we can show ORCH is an orchestrator generated by the OG for TS_t and $TS_{\mathcal{C}}$:

- by definition, $\text{ORCH}(\varepsilon) = \bot$;

- by defining $\sigma_r^{\text{ORCH}}(\varepsilon) = s_r^0$ and, for all $x{\cdot}a \in \mathcal{T}_t$, $\sigma_r^{\text{ORCH}}(x{\cdot}a) = \delta_r(\sigma_r^{\text{ORCH}}(x), a, \text{ORCH}(x{\cdot}a))$, we obtain that $\text{ORCH}(x{\cdot}a) \in \omega(\sigma_r^{\text{ORCH}}(x), a)$ for all $x{\cdot}a \in \mathcal{T}_t$. In facts, if we assume that there exists some $\bar{x}{\cdot}\bar{a} \in \mathcal{T}_t$ such that $\text{ORCH}(\bar{x}{\cdot}\bar{a}) \notin \omega_r(\sigma_r^{\text{ORCH}}(\bar{x}), \bar{a})$, then CLAB would not be a composition labeling, since there would exist no $k \in [1, \ldots, n]$ such that $\text{CLAB}(\bar{x}) = \langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ and $\text{CLAB}(\bar{x}{\cdot}\bar{a}) = \langle \bar{x}_1, \ldots, \bar{x}_k, \ldots, \bar{x}_n \rangle$.

Finally, we need to show that CLAB is defined with respect to ORCH, according to Definition 8, but this straightforward follows from requirement 2 of ORCH's construction. □

As already pointed out, given an orchestrator generator OG, Theorems 4 and 5 yield that by non-deterministically choosing, at each step, a service among those proposed by the selection function ω_r, we obtain all and only the orchestrators OG generates.

Interestingly, an orchestrator is not required to be built before a client starts interacting with the community, but it can be generated *just-in-time*, as client issues action requests, as shown next.

Definition 9 (Just-in-time orchestrator) *Given an orchestrator generator OG for TS_t and $TS_\mathcal{C}$, defined as in Definition 6, a* just-in-time (generated) orchestrator *is a function* $\text{JIT\_ORCH} : \mathcal{T}_t \rightarrow [1, \dots, n] \cup \bot$, *inductively defined as follows:*

- $\text{JIT\_ORCH}(\varepsilon) = \bot$;
- *if* $x{\cdot}a \in \mathcal{T}_t$, *then* $\text{JIT\_ORCH}(x{\cdot}a) = \mathtt{choose}(\omega_r(\sigma_r^{\text{JIT\_ORCH}}(x), a))$, *where* $\sigma_r^{\text{JIT\_ORCH}} : \mathcal{T}_t \rightarrow S_r$ *is the mapping function between nodes of $\mathcal{T}_t$ and corresponding states of S_r defined as in Definition 7 and* `choose` *stands for a choice function that chooses one element among those returned by* $\omega_r(\sigma_r^{\text{JIT\_ORCH}}(x), a)$.

Obviously, with appropriate choice functions for `choose`, one can get all possible generated orchestrators. But, the point of the definition above is that one can delay the choice performed by `choose` till run-time, where one can take into account information on the actual state, cost, etc., of the execution of actions by the various services. This gives a great flexibility to the orchestrator, which, in a sense, can "switch" composition on the go as needed. As a result, this work can be seen as providing formal bases for research work aimed at developing ambient-aware compositions that are fully reactive to events occurring during execution.

6. Conclusion

In this paper we have explored the possibility of basing service composition on the notion of simulation. We have seen that by using simulation, we are able to virtually compute all possible compositions at once, and that this opens the possibility of devising composition in a just-in-time fashion.

The tight connection between service composition and simulation discussed here, allows us to transfer well developed techniques for computing simulation, such as those in [16, 27, 10] to service composition.

Interestingly, also known result from service composition can be transferred to simulation. In particular, the EXPTIME-completeness of service composition studied here, allows us the say that checking simulation from a single deterministic transition system to the asynchronous product of to n deterministic transition systems is an EXPTIME-complete problem. Notably, this closes a long standing open problem in the simulation literature. Indeed, while in [26] the computational complexity characterization of checking simulation from the asynchronous product of n concurrent deterministic transition systems to a single deterministic transition system was given, the computational complexity characterization of checking simulation in the converse direction has remained open since. We close it here, by transferring EXPTIME-completeness result of service composition to simulation.

References

1. V. Agarwal, G. Chafle, K. Dasgupta, N. M. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *J. Web Sem.*, 3(4):311–339, 2005.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer, 2004.
3. D. Berardi. *Automatic Composition Services: Models, Techniques and Tools*. PhD thesis, Università degli Studi di Roma - La Sapienza, 2005.
4. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proc. of ICSOC 2003*, pages 43–58, 2003.
5. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(4):333 – 376, 2005.
6. D. Berardi, G. De Giacomo, and M. Mecella. Basis for automatic service composition. *Tutorial at WWW 2005*, 2005.
7. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. of WWW 2003*, 2003.
8. G. De Giacomo. Composition synthesis of web services composition synthesis of web services. *Talk at Daghstuhl Seminar on Synthesis and Planning*, 2005.
9. G. De Giacomo and M. Mecella. Service composition. *Tutorial at ICSOC 2004*, 2004.
10. R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: Coarsest partition problems. *J. Autom. Reasoning*, 31(1):73–103, 2003.
11. C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: lookaheads. In *Proc. of ICSOC 2004*, pages 252–262, 2004.
12. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kauffman, 2004.
13. G. Grahne and V. Kiricenko. Process mediation in an extended roman model. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*, pages 17 – 33, 2005.
14. A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.
15. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
16. M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of FOCS 1995*, pages 453–462, 1995.
17. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a Look Behind the Curtain. In *Proc. of PODS 2003*, pages 1–14, 2003.
18. M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proc. of PODS 2002*, pages 233–246, 2002.
19. S. A. McIlraith and T. C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. of KR 2002*, pages 482–496, 2002.
20. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web Services on the Semantic Web. *VLDB Journal*, 12(4):333–351, 2003.
21. M. Michalowski, J. L. Ambite, C. A. Knoblock, S. Minton, S. Thakkar, and R. Tuchinda. Retrieving and Semantically Integrating Heterogeneous Data from

the Web. *IEEE Intelligent Systems*, 19(3):72–79, 2004.

22. R. Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI 1971*, pages 481–489, 1971.
23. A. Muscholl and I. Walukiewicz. A Lower Bound on Web Services Composition. In *Proc. of the 10th Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS 2007)*, volume 4423 of *LNCS*. Springer, 2007.
24. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. of ICWS 2005*, 2005.
25. A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of POPL 1989*, pages 179–190, 1989.
26. S. K. Shukla, H. B. H. III, D. J. Rosenkrantz, and R. E. Stearns. On the complexity of relational problems for finite state processes. In *Proc. of ICALP 1996*, pages 466–477, 1996.
27. L. Tan and R. Cleaveland. Simulation revisited. In *Proc. of TACAS 2001*, pages 480–495, 2001.
28. W. Thomas. Infinite Games and Verification. In *Proc. of CAV 2002*, 2002.
29. J. D. Ullman. Information Integration using Logical Views. *Theoretical Computer Science*, 239(2):189–210, 2000.
30. M. Y. Vardi. An Automata-Theoretic Approach to Fair Realizability and Synthesis. In *Proc. of CAV 1995*, 1995.
31. J. Yang and M. P. Papazoglou. Service Components for Managing the Life-Cycle of Service Compositions. *Information Systems*, 29(2):97–125, 2004.