Service Composition in Stochastic Settings

Ronen I. Brafman[1], Giuseppe De Giacomo[2], Massimo Mecella[2], Sebastian Sardina[3]

[1] Ben-Gurion University, Beer-Sheva, Israel
brafman@cs.bgu.ac.il
[2] Sapienza Univ. Roma, Rome, Italy
{degiacomo,mecella}@dis.uniroma1.it
[3] RMIT University, Melbourne, Australia
sebastian.sardina@rmit.edu.au

Abstract. With the growth of the Internet-of-Things and online Web services, more services with more capabilities are available to us. The ability to generate new, more useful services from existing ones has been the focus of much research for over a decade. The goal is, given a specification of the behavior of the target service, to build a controller, known as an orchestrator, that uses existing services to satisfy the requirements of the target service. The model of services and requirements used in most work is that of a finite state machine. This implies that the specification can either be satisfied or not, with no middle ground. This is a major drawback, since often an exact solution cannot be obtained. In this paper we study a simple stochastic model for service composition: we annotate the target service with probabilities describing the likelihood of requesting each action in a state, and rewards for being able to execute actions. We show how to solve the resulting problem by solving a certain Markov Decision Process (MDP) derived from the service and requirement specifications. The solution to this MDP induces an orchestrator that coincides with the exact solution if a composition exists. Otherwise it provides an approximate solution that maximizes the expected sum of values of user requests that can be serviced. The model studied although simple shades light on composition in stochastic settings and indeed we discuss several possible extensions.

1 Introduction

With the growth of the Internet-of-Things (IoT) and online Web services, more and more services with more and more capabilities are available to us. By combining the functionalities offered by multiple services, we can provide much added value. A classic example is the ability to offer a complete vacation by combining Web services that offer (functionalities for buying) flights, ground transportation, accommodations, and event tickets. But as more physical devices are controlled through the Web via services, this can also be used to orchestrate the behavior of various kitchen devices, home entertainment systems, and home security services [1, 2].

The problem of service composition has been considered in the literature for over a decade, starting from seminal manual approaches, e.g., [3–5], which mainly focussed on modeling issues as well as on automated discovery of services described making use of rich ontologies, to automatic ones based on planning, e.g., [6, 7] or on KR techniques, e.g., [8], or on automated synthesis [9–11]. The reader interested in a survey of approaches can refer to [12, 13, 1]. Here we concentrate on the approach known in literature as the "Roman model" whose original paper [9] was awarded the most influential SOC paper of the decade prize at ICSOC 2013. Actually, for sake of simplicity, in our mathematical treatment we will consider the Roman model in its most pristine form. Though we will describe several extension in the discussion section.

In the Roman model, composition is as follows: each available (i.e., to be used in the composition, therefore referred to as *component*) service is modeled as a finite state machines (FSM), in which at each state, the service offers a certain set of actions, where each action changes the state of the service in some way. The designer is interested in generating a new service (referred to as *composite*, or *target*) from the set of existing services. The required service (the *requirement*) is specified using a FSM, too. The computational problem is to see whether the requirement can be satisfied by properly orchestrating the work of the component services. That is, by building a scheduler (called the *orchestrator*) that will use actions provided by existing services to implement action request of the requirement. Thus, a new service is synthesized using existing services.

Unfortunately, it is not always possible to synthesize a service that fully conforms with the requirement specification. Furthermore, a deterministic model (adopted in many approaches) is inappropriate for most services. Many services have various failure modes and different potential transitions for the same action. This can be addressed by allowing for non-determinism, but satisfying the requirement in this case can be even harder. This zero-one situation, where we can either synthesize a perfect solution or fail, should be improved. Rather than returning no answer, we need a notion of the "best-possible" solution, and the main contribution of this paper is to provide a solution to this problem.

In this paper we discuss and elaborate upon a probabilistic model for the service composition problem, first presented in [14]. In this model, an optimal solution can be found by solving an appropriate probabilistic planning problem (a Markov decision process – MDP) derived from the services and requirement specifications. Specifically, it is natural to make the requirement probabilistic, associating a probability with each action choice in each state. This probability captures how likely the user is to request the action in that state. Such information can be, initially, supplied by the designer, but can also be learned in the course of service operation in order to adapt the composition to user behavior. Next, a reward is associated with the requirement behavior. This reward can be defined in different ways depending on the designer's objectives. For example, we can associate a reward with different states that represent achieving certain milestones, so that solutions that make sure that the service is able to reach these milestones will be preferred. Or, we can associate a reward with actions at a state, modeling how important it is to provide the user with this option at this state. Thus, if certain actions represent crucial aspects of the service, they will be associated with high rewards, whereas actions that have added value, but are less important, can be associated with lower rewards.

We observe that rewards can be related to Quality-of-Service (QoS), which is often considered crucial in modeling Web services [15, 16]. Rewards on some states represent situations that the designer wants to enforce in order to guarantee QoS, while rewards on actions represent non-functional QoS requirements. As we discuss later on, one can use complex reward specifications in the form of transducers, or formulas in expressive logics such as LTL_f and LDL_f – linear-time temporal logic and dynamic logic on finite traces [17].

Given a set of available services and a probabilistic requirement specification, we formulate a new MDP that aggregates this information – it is very similar in spirit to the product automata used to solve the non-stochastic case – such that an optimal policy for this MDP generates an orchestrator that maximizes the expected sum of rewards. In some sense, the orchestrator will ensure that target transitions of highest value are provided for the longest possible time.

This model can also accommodate various useful extensions. For example, we can associate a cost with existing service actions or service states – e.g., energy use in the case of smart homes or service cost in the case of travel services. If these costs are commensurable with the value of services offered by the synthesized service, we still obtain a standard MDP. Otherwise, we obtain a multi-objective MDP (if we want to optimize both aspects) or a constrained MDP (if we have an energy or travel budget). Both models have been studied in literature and solution algorithms for them exist. In the last section, we discuss a number of such useful extensions.

Before continuing, we observe that our probabilistic extension to service composition is orthogonal to that proposed in [18], where available services are probabilistic, but the target specification (expressed as ω-regular languages, there) is not and the orchestrator is required to satisfy the target specification with probability 1.

The paper is structured as follows: Section 2 introduce our model of services, whereas Section 3 presents the model for the requirement and the solution of the proposed problem. In Section 5 we conlcude with a discussion some extensions of the basic framework.[4]

2 The Non-Stochastic Model

We adopt the Roman model for *service composition* [1], in its most pristine form [9], which we describe below. A *service* is defined as a tuple $S = (\Sigma, \sigma_0, F, A, \delta)$, where:

- Σ is the finite set of service's *states*;
- $\sigma_0 \in \Sigma$ is the *initial state*;
- $F \subseteq \Sigma$ is the set of service's *final* states;
- A is the finite set of service's *actions*;
- $\delta \subseteq \Sigma \times A \mapsto \Sigma$ is the service's *transition (partial) function*, i.e., *actions are deterministic*.

We interchange notations $s' \in \delta(\sigma, a)$ and $\sigma \xrightarrow{a} \sigma'$ in δ, possibly keeping implicit δ when no ambiguity arises. Finally, we write $A(\sigma)$ to denote $\{a \in A : \delta(\sigma, a) \text{ is defined}\}$ – the set of actions available at s.

[4] A preliminary version of this paper has been presented at the ICAPS 2017 Workshop on Generalized Planning. (The workshop does not have published proceedings.)

In the Roman model, we focus on the interface that services expose, which capture a *conversational* model of the service, i.e., one that represents the sequences of requests a service can serve, as the interaction with a client goes on. More specifically, from a given state, a service can serve only requests for actions that "label" an outgoing transition. Such actions, although atomic from the client perspective, correspond, in general to complex activities that may include, e.g., conversations with software modules or interactions with external users. Upon execution of the requested action, the service moves to a successor state, i.e., a state reachable from the current one via a transition labeled with the executed action.

A *history* h of a service S is a, possibly infinite, sequence alternating states and actions (necessarily ending with a state)

$$\sigma^0 \cdot a_1 \cdot \sigma^1 \cdot a_2 \cdot \cdots \cdot a_n \cdot \sigma^n \cdot \cdots$$

s.t. $\sigma^0 = \sigma_0$ and $\sigma^i \xrightarrow{a_{i+1}} \sigma^{i+1}$, for all $i \geq 0$. That is, a possible progression of the states of the service, annotated by an appropriate action. Note that the above implies that $a_i \in A(\sigma_{i-1})$.

We assume we have a finite set of available services $S_i = (\Sigma_i, \sigma_{i0}, F_i, A, \delta_i)$, over the same set of actions A. The set of all such services is referred to as the *service community*, denoted as $\mathcal{S} = \{S_1, \ldots, S_n\}$.

Given $\mathcal{S}$, [1] defines a *target* service as a further service $T = (\Sigma_t, \sigma_{t0}, F_t, A, \delta_t)$, again over the actions A. The target service provides a formal characterization of a desired service that may not be available in the community. We denote the set of possible target service histories by H_t.

Informally, the target represents a business process that one would like to offer to clients, where each state represents a decision point. At each state, the client is provided with a set of options to choose among, each corresponding to an action available in the state. Notice that typically the target service is not available. Further, the only entities able to execute actions, i.e., activities, are the available services. Thus, one cannot build the target service by simply combining the actions of the target service, but has to resort to the available services, which impose constraints on the execution of actions, depending on the conversations they can actually carry out.

The goal of service composition is to combine the available services in an appropriate way so as to mimic, from a client point of view, the behavior of the target service. This can be done by interposing an *orchestrator* between the available services and the client. The orchestrator delegates the current action requested by the client to some available service, waits for the service to fulfill it, then notifies the client, receives a new request, delegates it again, waits, and so on. In order to do this correctly, one not only needs to find a service that is able to execute the current action, but also has to choose the service so that *all possible future requests* compliant with the target service can be fulfilled.

To formally define the computational problem and its solution, we require some preliminary notions: The *system service* of $\mathcal{S}$ is the service $Z = (\Sigma_z, \sigma_{z_0}, F_z, A_z, \delta_z)$, s.t.:

- $\Sigma_z = \Sigma_1 \times \cdots \times \Sigma_n$;
- $\sigma_{z0} = (\sigma_{10}, \ldots, \sigma_{n0})$;

- $F_z = \{(\sigma_1, \ldots, \sigma_n) \mid \sigma_i \in F_i, 1 \leq i \leq n\}$
- $A_z = A \times \{1, \ldots, n\}$ is the set of pairs (a, i) formed by a shared action a and the index i of the service that executes it;
- $\sigma \xrightarrow{(a,i)} \sigma'$ iff, for $\sigma = (\sigma_1, \ldots, \sigma_n)$ and $\sigma' = (\sigma'_1, \ldots, \sigma'_n)$, it is the case that $\sigma_i \xrightarrow{a} \sigma'_i$ in δ_i, and $\sigma_j = \sigma'_j$, for $j \neq i$.

Intuitively, Z is the service stemming from the product of the asynchronous execution of the services in $\mathcal{S}$. This is a virtual entity, i.e., without any actual counterpart, that offers a formal account of the evolution of the available services, when the community is seen as a whole. Note that in the transitions of Z, the service executing the corresponding action, is explicitly mentioned. Also $(a, i) \in A(\sigma_z)$ indicates that a can be executed by service i in the current state. We denote the set of system service histories by H_z.

An *orchestrator* for a community $\mathcal{S}$ is a partial function[5]:

$$\gamma : \Sigma_z \times A \mapsto \{1, \ldots, n\}.$$

Intuitively, γ is a decision maker able to keep track of the way the services in $\mathcal{S}$ have evolved up to a certain point, and that, in response to an incoming action request, returns the index of a service.

Notice that, in general, γ is not guaranteed to return a service able to execute the requested action, nor that delegating the action to the returned service guarantees that all possible future requests can be served. Obviously, only the orchestrator that guarantees such features can be actually used to realize the desired service, as formalized below.

The dynamics of the system is deterministic given the actions selected by the user. Hence, together with the orchestrator choice, it determines a system history. That is, an orchestrator defines a partial function from target-service histories to system histories, based on the (partial) mapping from system state and action to a service and the (partial) mapping from system state, action, and service, to the next system state. We denote this mapping by τ. More formally, $\tau : H_t \mapsto H_z$ is defined inductively as follows: $\tau(\sigma_{t0}) = \sigma_{z0}$. Let $\tau(h_t) = h_z$, and let s_t, s_z denote the last states, respectively, in h_t, h_z. Then, τ is also defined on $h_t \cdot a \cdot s'_t$ provided: $a \in A(s_t)$ and $s'_t = \delta_t(s_t, a)$, and that γ is defined on (s_z, a), and $(a, \gamma(s_z, a)) \in A(s_z)$. That is, provided the orchestrator function is defined on s_z and a, assigning some value i, and δ_z is well defined on (s_z, i), we have $\tau(h_t \cdot a \cdot s'_t) = h_z \cdot a \cdot \delta_z(s_z, (a, i))$. Otherwise, $\tau(h_t \cdot a \cdot s'_t)$ is undefined.

If $\tau(h_t)$ is well defined, we say that target history h_t is *realizable* by the orchestrator.

The orchestrator γ is said to *realize* a target service Z if it realizes all histories of Z. In this case, γ is also called a *composition* of Z (on $\mathcal{S}$).

The problem of service composition in known to be EXPTIME-complete, in fact exponential on the number of the available services [9, 19] and techniques based on model checking, simulation, and LTL synthesis are available [1]. Also, several variants have been studied, including the case of nondeteministic (i.e., partially controllable but fully observable) available services [11].

[5] In the original orchestrator definition γ is a function of the entire history instead of the system service's current state only. It can be shown that if an orchestrator of the previous form exist then one of the current form exists [9, 11]. So we adopt this simpler notion.

3 The Valued Requirement Model

The main limitation of the composition approach outlined above is that if a composition does not exists, no notion of a "good" or "approximate" solution exists. An interesting notion of unique supremal composition has been introduced in [20]. But this notion puts the burden on the client executing the target to foresee in advance what requests it will ask in the future, and this may be too limiting in various contexts. Furthermore, in actual applications, requests are not usually of uniform importance. Some parts of the target service may be good to have, but not essential, while other parts may be central to its functionality. And typically, different requests are not equally likely. These considerations are not captured by the above model and its solution concepts. Hence, we propose a modified model that takes these considerations into account, thus obtaining a richer, finer grained, formulation of the objective that allows us to define appealing notions of "optimal" compositions.

To model the value and likelihood of requets, we augment the target service model with two additional elements. P_t will be a distribution over the actions given the state. $P_t(s, a)$ is the likelihood that a user will request a in target state s. Technically, $P_t(s)$ returns a distribution over the actions, or the empty set, when s is a terminal state on which no actions are possible. R_t is the reward function, associating a non-negative reward with the ability to provide the action requested by a user. $R_t(s, a)$ is the value we associate with being able to provide action a in state s. Formally, a target service is $T = (\Sigma_t, \sigma_{t0}, F_t, A, \delta_t, P_t, R_t)$, where $\Sigma_t, \sigma_{t0}, F_t, A, \delta_t$ are defined as before, $P_t : \Sigma_t \to \pi(A) \cup \emptyset$ is the action distribution function, and $R_t : \Sigma_t \times A \to \mathbb{R}$ is the reward function. We assume rewards are non-negative.

One can specialize this definition in various ways: R_t can depend on Σ_t only, if for example, we assume that the reward is given for reaching a final state, or some particular "normal" finite states, capturing the fact that the service has completed appropriately. R_t could simply assign an identical positive value to every pair (σ, a) such that $a \in A(\sigma)$. This essentially implies that what we care about is the ability to service as many actions as possible in a state.

The definitions of an orchestrator, a target history, a realizable target history, and a realizable target do not change. But we can now define additional notions. First, P_t induces a probability density function over the set of all infinite target histories, which we will denote by P_∞. (This follows by the Ionescu Tulcea extension theorem.) Second, R_t can be used to associate a value with every infinite history. The standard definition of the value of a history h_t, which we adopt here, is that of the sum of discounted rewards: $v(\sigma_0, a_1, \sigma_1, \cdots) = \sum_{i=0}^{\infty} \lambda^i R_t(\sigma_i, a_{i+1})$, where $0 < \lambda < 1$ is the discount factor. The discount factor can be viewed as measuring the factor by which the value of rewards is reduced as time progresses, capturing the intuition that the same reward now is better than in the future.[6] [7]

[6] It can also be viewed as quantifying the probability $(1 - \lambda)$ that the process will terminate at some state.

[7] An alternative notion, for which similar results can be obtained is that of average reward, defined, e.g., as $\liminf_{m\to\infty} \frac{1}{m} \sum_{i=0}^{m} R_t(\sigma_i, a_{i+1})$, which requires more mathematical sophistication to handle.

Given the above, we can define the expected value of an orchestrator γ to be:

$$v(\gamma) = E_{h_t \sim P_\infty}(v(h_t) \cdot real(\gamma, h_t))$$

where $real(\gamma, h_t)$ is 1 if h_t is realizable in γ, and 0 otherwise. That is, $v(\gamma)$ is the expected value of histories realizable in γ. Finally, we define an *optimal* orchestrator to be $\gamma = \arg\max_{\text{orchestrator } \gamma'} v(\gamma')$. The following is reassuring:

Theorem 1. *If the target is realizable and every target history has strictly positive value then γ realizes the target iff it is an optimal orchestrator.*

That is, if it is possible to realize the target requirement, then any orchestrator realizing it is optimal, and any orchestrator that does not realize some history, is non-optimal. The former stems from the fact that if the set of histories realizable using orchestrator γ contains the set realizable using orchestrator γ', then $v(\gamma) \geq v(\gamma')$. The latter stems from the fact that if, in addition, the set of histories realizable by γ but not by γ' has positive probability, then $v(\gamma) > v(\gamma')$. Now, if h is not realizable by γ', there exists a point in h where γ' does not assign the required action to a service that can supply it. Thus, any history that extends the corresponding prefix of h is not realisable, and the set of such histories has non-zero probability. Since we assume all histories have positive value, we obtain the desired result.

The importance of this new model is that we now have a clear notion of an optimal orchestrator that works even when the target service is not fully realizable, and this notion is clearly an extension of the standard notion, coinciding with it when the service is realizable by some orchestrator. An optimal controller is simply one that is able to handle more (in expectation) valued histories.

4 Computing an Optimal Orchestrator

We now explain how to solve the above model by formulating an appropriate MDP. An MDP is a four-tuple $M' = (S', A', Tr', R')$, where S' is a finite set of states, A' a finite set of actions, $Tr' : S' \times A' \to \pi(S')$ is the transition function, and $R : S' \times A' \to \mathbb{R}$ is the reward function. The two latter terms were defined above in the context of the valued composition model.

The composition MDP is a function of the system service and the target service as follows $M(Z, T) = (S_M, A_M, Tr_M, R_M)$, where *(i)* $S_M = \Sigma_Z \times \Sigma_T \times A \cup s_{M0}$ *(ii)* $A_M = \{a_{M0}, 1, \ldots, n\}$ *(iii)* $Tr_M(s_{M0}, a_{M0}, (\sigma_{z0}, \sigma_{t0}, a)) = P_t(\sigma_{t0}, a)$ *(iv)* $Tr_M((\sigma_z, \sigma_t, a), i, (\sigma'_z, \sigma'_t, a')) = P_t(\sigma'_t, a')$ if $\sigma_z \xrightarrow{(a,i)} \sigma'_z$ and $\sigma_t \xrightarrow{a} \sigma'_t$, and 0 otherwise. *(v)* $R((\sigma_z, \sigma_t, a), i) = R_t(\sigma_s, a)$ if $(a, i) \in A(\sigma_z)$ and 0 otherwise.

That is, the set of states is the product of the states of the system service, the states of the target service, and the set of actions. Intuitively, the state (σ_z, σ_t, a) denotes the fact that the system state is currently σ_z, the target state is currently σ_t and the requested action is a. In addition, there is a distinguished initial state s_{M0}. The actions correspond to selecting the service that will provide the current requested actions, together with a special initializing action, a_{M0}. A transition in state s_{M0} is defined only for action a_{M0}. From this state, we can get to state $(\sigma_{z0}, \sigma_{t0}, a)$ with probability that is equal to the probability that action a would be requested from the target service at its initial state. The state $(\sigma_{z0}, \sigma_{t0}, a)$ represents the situation that the system and target service are in their initial state, and that a is requested of the target service. In general, the defintion of $Tr((\sigma_z, \sigma_t, a), i, (\sigma'_z, \sigma'_t, a'))$ captures the fact that if service S_i provides action a in system and target states σ_z and σ_t, then the next system state is determined by (a, i) and the previous system state, and the next target state is determined by a and the previous target state. The probability associated with this transition is the probability that action a' will be requested in the new target state. Finally, the reward function associates a positive reward with states in which the assigned service S_i is able to perform the requested action a, and the value of this reward is the value of doing actions a at the target state.

Theorem 2. *Let ρ be an optimal policy for $M(Z, T)$. Then, the orchestrator γ such that $\gamma((\sigma_z, \sigma_t), a) = \rho(\sigma_z, \sigma_t, a)$ is an optimal orchestrator.*

Above we assume that an optimal policy for the MDP is one maximizing expected discounted sum of rewards with discount factor λ. The result follows from the fact that there is a one-to-one correspondence between orchestrators and policies for $M(Z, T)$, via the relationship: $\gamma((\sigma_z, \sigma_t), a) = \rho(\sigma_z, \sigma_t, a)$, and the fact that the value of policy ρ so defined equals $v(\gamma)$.

5 Extensions

With the basic setting indtroduced, here we can now discuss several possible extensions.

Stochastic available services. For the sake of simplicity, we have assumed so far that the component and target services are deterministic. Extending our model to capture stochastic services, where the service transitions are probabilistic too, is quite easy, see [14]. One needs to simply alter the relevant transition functions. The precise definition of realizability now becomes slightly more cumbersome to write, but the underlying intuitions are the same. The MDP construction, too, need only be modified slightly to take into account the stochastic transitions of the system state and target state.

Handling exceptions. Our current model does not explicitly capture a critical aspect of many real-world scenarios, exception handling [21]: if the target/composite service terminates before a terminal state has been reached, work done so far has to be undone. This work is distributed across different services. For example, if while booking a vacation, we book a flight but cannot book a hotel, we must cancel the flight reservation, which can be costly. If we also booked a car by now, the cost would be higher. We can augment the MDP defined earlier to take these costs into account by adding a negative reward to states (s_z, s_t, a) and service choice i such that i cannot supply action a in its current state. The size of the reward can depend on the states of the various services, as reflected in s_z, which reflects the work that needs to be undone in each of the existing services.

Separate rewards specifications. In the setting considered here, we have coupled the rewards with the likelihood of the client making certain action requests into the target service to be realized. In fact it may be convenient to keep the two specification separated, and use the target service only to specify the likelihood of action request, in line with what happens in the deterministic case. Rewards in this case could be expressed dynamically on the history of actions executed so far by the target, through a transducer.

More precisely a *transducer* $R = (\Sigma, \Delta, S, s_0, f, g)$ is a deterministic transition system with inputs and outputs, where Σ is the input alphabet, Δ is the output alphabet, S is the set of states, s_0 the initial state, $f : S \times \Sigma \longrightarrow S$ is the transition function (which takes a state and an input symbol and returns the successor state) and $g : S \times \Sigma \longrightarrow \Delta$ is the output function (which returns the output of the transition).

In our case the input alphabet would be the set of actions A, the output alphabet the possible rewards expressed as reals $\mathbb{R}$. In this way the output function $g : S \times A \longrightarrow \mathbb{R}$, would correspond the *reward function*. The point is that now the rewards do not depend on the state of the target, but on the sequence of actions executed so far. Interestingly if we take the synchronous product of the target T (without rewards, but with stochastic transitions) and of R (which is deterministic but outputs rewards), we get a target of the form specified in Section 3, though this time computed from the two separated specifications, and we can apply the MDP construction presented here (or its extension with stochastic available services discussed previously).

Non-Markovian rewards. In line with the above point, it has long been observed [22, 23] that many performance criteria call for more sophisticated reward functions that do not depend on the last state only.

For example, in Robotics [24], we may want to reward a robot for picking up a cup only if it was requested to do so earlier, where the pick-up command may have been given a number of steps earlier. Similarly, we may want to reward an agent for behavior that is conditional on some past fact – for example, if the person was identified as a child earlier, we must provide her with food rich in protein, and if he is older, in food low in sodium. Or we may want to reward the robot for following some rules, such as executing an even number of steps back and forth, so as to end up in the starting position.

All these proposal share the idea of specifying rewards on (partial traces or histories) through some variant of linear-time temporal logic over finite traces LTL$_f$. The research on variants of LTL$_f$ has become very lively lately with promising results [17, 25–28]. A key point is that formulas in these logics can be "translated" into standard deterministic finite state automata DFAs that recognize exactly the traces that fulfill the formula. Such DFAs can be combined with probabilistic transition systems to generate suitable MDPs to be used for generating optimal solutions. This can be done also in our context. Essentially we replace (or enhance) the target specification with a declarative set of logical constraints. Then we compute the synchronous product with a target transition system that us the likelihood of action choice, hence getting a target specification as that of Section 3, analogously to the case of the transducer above. This can be solved by the techniques presented earlier.

High-level programs as target services. Often certain non-Markovian specifications can be expressed more naturally by using procedural constraints [25, 29, 30]. In particular, we can introduce a sort of propositional variant of GOLOG [31]:

$$\begin{array}{l} \delta ::= A \mid \varphi? \mid \delta_1 + \delta_2 \mid \delta_1 ; \delta_2 \mid \delta^* \mid \\ \qquad \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \ \mid \textbf{while } \phi \textbf{ do } \delta \end{array}$$

Hence, we can assign rewards to traces that correspond to successful computations of such programs.

For example in a smart environment such as that in [32] we could have a reward associated to completing the following program:

```
while(true) do
    if (cold ∧ windowOpen)) then
        closeWindow;
        turnOnFirePlace + turnOnHeating
```

which says that, all along, if it is cold and the window is open, then immediately close the window and either turn on the fire place or the heating system (no other actions can interleave this sequence).

Note that **if** and **while** can be seen as abbreviations for regular expression [33], namely:

$$\begin{array}{r} \textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \ \doteq (\phi?; \delta_1) + (\neg\phi?; \delta_2) \\ \textbf{while } \phi \textbf{ do } \delta \doteq \ (\phi?; \delta)^*; \neg\phi? \end{array}$$

Hence these programs can also be translated into *regular expressions* and hence in DFA to be used as above.

Interestingly, we can combine procedural and declarative temporal constraints by adopting a variant of LTL$_f$, called linear-time dynamic logic on finite traces, or LDL$_f$, as specification language [17]. For example we may write

$$\begin{array}{l} [\mathit{true}^*]\langle \textbf{while}(\mathit{cold} \wedge \mathit{heatingOn})) \textbf{ do} \\ \qquad\qquad (\neg \mathit{turnOffHeating}^*; \mathit{heat})\rangle \end{array}$$

which says that at every point in time, while it is cold and the heating is on then heat, possibly allowing other action except turning off heating. Again we are able to transform these formulas into DFAs and proceed as discussed above.

Finally these ideas are related to so called agent planning programs [34], where the target is specified as a network of declarative goals. Such programs can also be extended to the stochastic setting presented here

Learning. Although we focus on this paper on model specification and model-based solution techniques, we point out that for Web services, statistics gathering is very simple, and in fact, is carried out routinely nowadays. Consequently, it is not difficult to learn the stochastic transition function of existing services online, and use it to specify the probabilistic elements of the model.

6 Conclusion

In the service composition problem, we attempt to satisfy a specification of a new service using existing services. This allows users and businesses to define new, complicated services with added value on top of an existing set of services. By improving the Roman model of service composition to include request likelihood and values, we were able to not only provide a more faithful formal model of the problem, but also to address the long-standing problem of defining optimal orchestrators when no orchestrator can realize all possible desirable behaviors. Thanks to the correspondence established between orchestrators and policies of a suitably defined MDP, we can also show how such orchestrators can be easily computed. Moreover the setting proposed can be extended in several directions of great theoretical and practical interest.

References

1. De Giacomo, G., Mecella, M., Patrizi, F.: Automated service composition based on behaviors: The roman model. In: Web Services Foundations. Springer (2014) 189–214
2. Bronsted, J., Hansen, K.M., Ingstrup, M.: Service composition issues in pervasive computing. IEEE Pervasive **9**(1) (2010)
3. Medjahed, B., Bouguettaya, A., Elmagarmid, A.: Composing web services on the semantic web. Very Large Data Base Journal **12**(4) (2003) 333–351
4. Yang, J., Papazoglou, M.: Service components for managing the life-cycle of service compositions. Information Systems **29**(2) (2004) 97–125
5. Cardoso, J., Sheth, A.: Introduction to semantic web services and web process composition. In: Proc. 1st Int. Work. on Semantic Web Services and Web Process Composition (SWSWPC). (2004)
6. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: Automating daml-s web services composition using shop2. In: ISWC. (2003)
7. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated composition of Web services by planning at the knowledge level. In: IJCAI. (2005)
8. McIlraith, S., Son, T.: Adapting golog for composition of semantic web services. In: KR. (2002)
9. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: ICSOC. (2003)
10. Hu, Y., De Giacomo, G.: A generic technique for synthesizing bounded finite-state controllers. In: ICAPS. (2013)
11. De Giacomo, G., Patrizi, F., Sardiña, S.: Automatic behavior composition synthesis. Artif. Intell. **196** (2013) 106–142
12. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: OTM. (2008)
13. Su, J.: Semantic web services: Composition and analysis. IEEE Data Engineering Bulletin **31**(3) (2008)
14. Yadav, N., Sardiña, S.: Decision theoretic behavior composition. In: AAMAS. (2011)
15. Menasce, D.: Qos issues in web services. IEEE Internet Computing **6**(6) (2002)
16. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Transactions on Software Engineering **30**(5) (2004)
17. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. (2013)

18. Nain, S., Lustig, Y., Vardi, M.Y.: Synthesis from probabilistic components. Logical Methods in Computer Science **10**(2) (2014)
19. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. Logical Methods in Computer Science **4**(2) (2008)
20. Yadav, N., Felli, P., De Giacomo, G., Sardiña, S.: Supremal realizability of behaviors with uncontrollable exogenous events. In: IJCAI. (2013) 1176–1182
21. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: AIMSA. (2004)
22. Bacchus, F., Boutilier, C., Grove, A.J.: Rewarding behaviors. In: AAAI. (1996)
23. Thiébaux, S., Gretton, C., Slaney, J.K., Price, D., Kabanza, F.: Decision-theoretic planning with non-markovian rewards. J. Artif. Intell. Res. **25** (2006) 17–74
24. Lacerda, B., Parker, D., Hawes, N.: Optimal policy generation for partially satisfiable co-safe LTL specifications. In: IJCAI. (2015)
25. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: IJCAI. (2015)
26. De Giacomo, G., Vardi, M.Y.: $\mathrm{LTL_f}$ and $\mathrm{LDL_f}$ synthesis under partial observability. In: IJCAI. (2016)
27. Torres, J., Baier, J.A.: Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In: IJCAI. (2015)
28. Camacho, A., Triantafillou, E., Muise, C., Baier, J.A., McIlraith, S.: Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In: AAAI. (2017)
29. Fritz, C., McIlraith, S.A.: Monitoring plan optimality during execution. In: ICAPS. (2007)
30. Baier, J.A., Fritz, C., Bienvenu, M., McIlraith, S.A.: Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In: AAAI. (2008)
31. Levesque, H.J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. J. of Logic Programming **31** (1997)
32. De Giacomo, G., Di Ciccio, C., Felli, P., Hu, Y., Mecella, M.: Goal-based composition of stateful services for smart homes. In: OTM. (2012)
33. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. of Computer and System Sciences **18** (1979)
34. De Giacomo, G., Gerevini, A.E., Patrizi, F., Saetti, A., Sardiña, S.: Agent planning programs. Artif. Intell. **231** (2016)