Adaptive Service Composition Based on Reinforcement Learning*

Hongbing Wang[1], Xuan Zhou[2], Xiang Zhou[1], Weihong Liu[1], Wenya Li[1], and Athman Bouguettaya[2]

[1] School of Computer Science and Engineering, Southeast University, China
{hbw,szs,chw}@seu.edu.cn
[2] CSIRO ICT Centre, Australia
{xuan.zhou,athman.bouguettaya}@csiro.au

Abstract. The services on the Internet are evolving. The various properties of the services, such as their prices and performance, keep changing. To ensure user satisfaction in the long run, it is desirable that a service composition can automatically adapt to these changes. To this end, we propose a mechanism for adaptive service composition. The mechanism requires no prior knowledge about services' quality, while being able to achieve the optimal composition solution by leveraging the technology of reinforcement learning. In addition, it allows a composite service to dynamically adjust itself to fit a varying environment, where the properties of the component services continue changing. We present the design of our mechanism, and demonstrate its effectiveness through an extensive experimental evaluation.

1 Introduction

In the emerging paradigm of Service Oriented Computing (SOC), data, software and hardware can all be encapsulated as services shared on the Internet. Applications would no longer be built from scratch, but as compositions of the available services. In this way, application builders can focus on business logics, without overly spending time and efforts on infrastructures. To realize SOC, a variety of technologies have been proposed. They include the stack of Web Service technologies, e.g. SOAP and BPEL, and a variety of mashup tools, e.g. Yahoo Pipe and Google Mashup Editor. Most of these technologies aim to help engineers / users create service compositions efficiently.

The services on the Internet keep evolving. Some services may stop functioning, once their providers go out of business. Some may keep upgrading themselves, to achieve improved Quality of Service (QoS). For instance, Amazon EC2 has decreased its prices for several times (the latest one was a 15% cut in Nov 2009), but continually improve its architecture to achieve better performance.

* This work is partially supported by the NSFC project No. 60673175 and the Jiangsu NSF project titled "Cloud-Service Oriented Autonomic Software Development".

P.P. Maglio et al. (Eds.): ICSOC 2010, LNCS 6470, pp. 92–107, 2010.

For those services that do not evolve, their quality may change with the varying environment. For instance, the growth of customers will usually increase the response time of a service. Due to the evolvement of services and the dynamicity of the environment, a service composition has to be continually adjusted, to keep functioning in a cost-effective manner. This imposes intensive workload to engineers in monitoring, tuning and re-engineering service compositions. It is desirable that a service composition can be self-adaptive in a dynamic environment, so that it will incur less maintenance cost.

In recent years, extensive research efforts have been spent on the development and the optimization of service compositions [13,19]. However, most of the existing approaches assume a static environment. As a common practice [19,1,18], an abstract service composition is firstly created to meet users' requirements on functionality. Then, concrete services are selected based on their non-functional properties to create a concrete service composition of the best possible quality. A service composition created out of this approach runs in a static workflow and is not self-adaptive. When its environment or its component services change, the composition has to be manually adjusted to adapt to the changes.

In this paper, we present a novel mechanism to enable a service composition to adapt to its environment autonomously. Our mechanism achieves self-adaptivity by utilizing Reinforcement Learning, a typical technology used for planning and optimization in dynamic environments. We model a service composition as a Markov Decision Process (MDP), so that multiple alternative services and workflows can be incorporated into a single service composition. The optimization of the composition is conducted at runtime (when users consume the services), through reinforcement learning. The learning aims to obtain the optimal policy of the Markov decision process that delivers the best quality of service. In contrast to the existing approaches of service composition and optimization, our mechanism requires no prior knowledge of services' non-functional properties, which are anyway uncertain in most real world circumstances. Instead, it learns these properties by actually executing the services. As the learning process continues throughout the life-cycle of a service composition, the composition can automatically adapt to the change of the environment and the evolvement of its component services. We have conducted extensive experiments to evaluate our approach and observed a number of its merits.

The remainder of this paper is organized as follows. Section 2 defines our model of service composition. Section 3 shows how reinforcement learning can be conducted to run the service compositions. Section 4 presents the results of our experimental evaluation. Section 5 compares our approach against some related work. Finally, Section 6 provides a conclusion.

2 A MDP Model for Service Composition

Some recent approaches [5,6] to automatic service composition have used Markov Decision Process (MDP) to model service compositions. We use this model too, as it allows us to apply reinforcement learning to dynamically optimize the quality of a service composition.

We first define the key concepts of the model.

Definition 1 (Web Service). A Web service is modeled as a tuple $WS =< ID, QoS >$, where
- ID is the identifier of the Web service.
- QoS is a n-tuple $< att_1; att_2; ...; att_n >$, where each att_i denotes a QoS attribute of WS. ■

Example 1: Some example Web services are defined as follows:

ID: StorkID
QoS: <price:free, response_time:1ms, availability:99%>

ID: StorkPrice
QoS: <price:free, response_time:1ms, availability:99%>

ID: StrokInfo1
QoS: <price:$1, response_time:200ms, availability:80%>

ID: StockInfo2
QoS: <price:$2, response_time:1ms, availability:99%>

ID: Transaction
QoS: <price:$1, response_time:200ms, availability:99%>

We use these services as walk-through examples to illustrate our approach. Among these services, StockID allows a user to know the identification code of a stock. StockPrice allows a user to check the current price of a stock. StockInfo1 and StockInfo2 are two alternative services for providing user with more information about stocks. They charge different fees and offer different QoS. Finally, Transaction allows a user to buy or sell stock. In our example, we use only three typical QoS attributes. ■

As we use Markov Decision Process (MDP) to model service composition, we first define MDP. In [8], a MDP is defined as:

Definition 2 (Markov Decision Process (MDP)). A MDP is a 4-tuple $M =< S, A(.), P, R >$, where
- S: a finite set of states of the world. When an agent arrives at a state, the agent can observe the complete state of the world.
- $A(s)$: a finite set of actions. The set of available actions depends on the current state $s \in S$.
- P: when an action $a \in A$ is performed, the world makes a probabilistic transition from its current state s to a resulting state s' according to a probability distribution $P(s'|s, a)$.
- R: Similarly, when action a is performed and the world makes its transition from s to s' , the agent receives a real-valued (possibly stochastic) reward r, whose expected value is $r = R(s'|s, a)$. ■

A MDP involves multiple actions and paths for a agent to choose. By using it to model service compositions, we are able to integrated multiple alternative workflows and services into a single composition. We call our model of service composition WSC-MDP, which simply replaces the actions in a MDP with Web services.

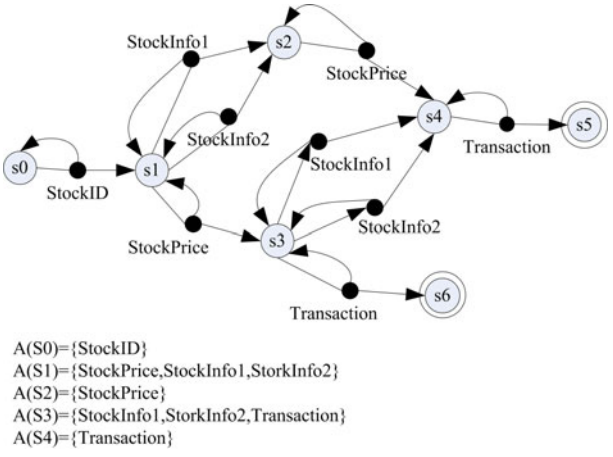


Fig. 1. The WSC-MDP of a Composite Service for Stock Transaction

Definition 3 (Web Service Composition MDP (WSC-MDP)). A Web service composition MDP is 6-tuple WSC-MDP$=< S, s_0, S_r, A(.), P, R >$, where

- S: a finite set of states of the world.
- $s_0 \in S$is the initial state. An execution of the service composition starts from this state.
- $S_r \subset S$ is the set of terminal states. Upon arriving at one of the states, an execution of the service composition terminates.
- $A(s)$ represents the set of Web services that can be executed in state $s \in S$. A service ws belongs to $A(s)$, only if the precondition ws^P is satisfied by s.
- P: When a Web service $ws \in A(s)$ is invoked , the world makes a transition from its current state s to a resulting state s', where the effect of ws is satisfied. For each s, the transition occurs with a probability $P(s'|s, ws)$.
- R: When a Web service $ws \in A(s)$ is invoked, the environment makes a transition from s to s', and the service consumer receives an immediate reward r, whose expected value is $R(s'|s, ws)$. ■

A WSC-MDP can be visualized as a transition graph [15]. As illustrated by Fig. 1, the graph contains two kinds of nodes, i.e., state nodes and service nodes, which are represented by open circles and solid circles respectively. $s0$ represents the initial state node. The terminal states nodes are those with double circles. A state node can be followed by a number of service nodes, representing the possible services that can be invoked in the state. There is at least one arrow pointing from a service node to the next state node. Each arrow is labeled with a transition probability $P(s'|s, ws)$, and the expected reward for that transition $R(s'|s, ws)$, which are determined by the P and R in the WSC-MDP. (For simplicity, we omit the labels in Fig. 1.) The transition probabilities on the arrows rooted at a single action node always sum to one.

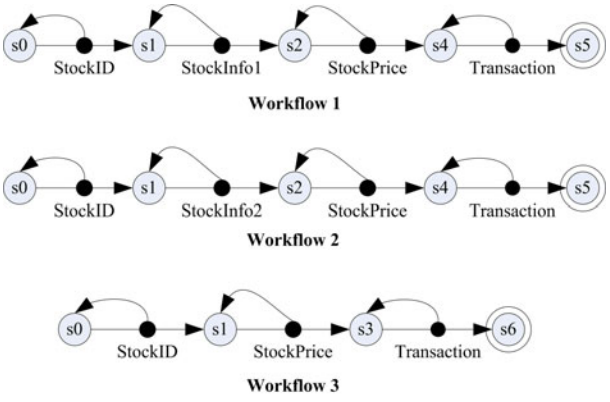


Fig. 2. Three Workflows contained by the WSC-MDP in Fig. 1

Example 2: The services introduced in Example 1 can be composed into a integrated service for stock transaction. We model it as a WSC-MDP transition graph in Fig. 1. (For simplicity, we assume only two resulting states for each service. If the service is invoked successfully, the world changes to the next state predicated by the service's post-condition. Otherwise, the world remains in the current state.) This service composition provides multiple workflows for users to purchase or sell stocks. When executing the composition, the system can choose the workflow that offers the best results. ■

As shown in the related work [5,6], WSC-MDP is expressive enough to describe the control flows of a general business process. In addition, WSC-MDP can express a super service composition composed of multiple alternative workflows. Each of the workflows corresponds to a service composition constructed by traditional approaches, such as BPEL.

Definition 4 (Service Workflow). Let wf be a subgraph of a WSC-MDP. wf is a service workflow if and only if there is at most one service that can be invoked at each state wf. In other words, $\forall s \in wf, |A(s) \cap wf| \leq 1$. ■

A service workflow is actually analogous to a deterministic state machine. A tradition service composition usually corresponds a single workflow. In contrast, a WSC-MDP based composition can consist of multiple workflows.

Example 3: The WSC-MDP in Example 2 consists of multiple service workflows. Fig. 2 shows three of them. Which workflow will be executed by the service composition is determined the policy of the Markov decision process. ■

We define a policy (or execution policy) of a WSC-MDP as follows.

Definition 5 (Policy). A policy π is a mapping from state $s \in S$ to a service $ws \in A$, which tells which service $ws = \pi(s)$ to execute when the world is in state s. ■

Example 4: A policy for the WSC-MDP in Example 2 can be expressed as $\pi = \{s0 : StockID, s1 : StockInfo1, s2 : StockPrice, s4 : Transaction\}$. It tells our system to execute the service composition using Workflow 1 in Fig. 2(a). ■

Each policy can uniquely determine a workflow of a WSC-MDP. By executing a workflow, the service customer is supposed to receive a certain amount of reward, which is equivalent to the cumulative reward of all the executed services. Given a WSC-MDP, the task of our service composition system is to identify the optimal policy or workflow that offers the best cumulative reward. As the environment of a service composition keep changing, the transition function P and the reward function R of a WSC-MDP changes too. As a result, the optimal policy changes with time. If our system is able to identify the optimal policy at any given time, the service composition will be highly adaptive to the environment.

It worth noting that this paper only deals with how to model a service composition using WSC-MDP. Regarding the design and construction of WSC-MDP, we schedule it in our future work. In practice, a WSC-MDP can either be created manually by engineers or using some automatic composition approaches, such as an AI planner [11,14].

3 Reinforcement Learning for Service Composition

The MDP model introduced previously allows service engineers to integrate multiple alternative workflows and services into a single service composition. During the execution of a service composition, the system can dynamically choose the optimal policy / workflow that would give the best possible reward to the users. When the complete WSC-MDP is known, the theoretically optimal policy can always be calculated. However, this is not true in practice. Firstly, we may not have complete knowledge about the state transition functions of the WSC-MDP, as the results of a service are not always predicable. Secondly, we may not have sufficient knowledge about the reward functions of the WSC-MDP. Especially for human oriented services, the user experience of a service composition is rarely predicable, until it has been tried out. Moreover, as the environment of a service composition keep changing, both the state transition functions and the reward functions change with time.

Due to the above issues, we choose to learn the optimal policy of a WSC-MDP at runtime. In this section, we introduce a reinforcement learning scheme to orchestrate WSC-MDP based service compositions. We first give a brief overview of a reinforcement learning algorithm called Q-Learning. Following that, we show how to apply reinforcement learning to WSC-MDP.

3.1 Q-Learning

In reinforcement learning, the task of the learner or decision-maker is to learn a policy of the MDP that maximizes the expected sum of reward. As there is no initial and terminal states in a generic MDP (Definition 2), an agent is supposed

to live in the MDP forever. Therefore, the cumulative reward of starting from an arbitrary state s_t and following a policy π is defined as:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{1}$$

where r_{t+i} is the *expected* reward received in each step, and γ is a discount factor.

Based on Equation 1, the optimal policy is the policy that maximizes $V^{\pi}(s_t)$ for all s_t. Let π^* denote the optimal policy. Therefore,

$$\pi^* = argmax_{\pi} V^{\pi}(s_t), (\forall s_t \in S) \tag{2}$$

We use $V^*(.)$ to represent the cumulative reward of the optimal policy π^*.

As mentioned earlier, the state transition function and reward function of a MDP may not be known. Thus, π^* cannot be calculated directly. It has to be learned through a trial-and-error process.

To facilitate the learning process, Q-Learning [17] uses a Q function to simulate the cumulative reward. Let s be the current state of the agent. Let a be the action taken by the agent. Let s' be the resulting state of action a. Then, the Q function of taking action a at state s is

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma V^*(s')] \tag{3}$$

The Q function represents the best possible cumulative reward of taking action a at state s.

Based on Equation 3, we obtain the optimal policy for each single state s.

$$\pi^*(s) = argmax_a Q(s,a), (\forall a \in A(s)) \tag{4}$$

Applying Equation 4 to resolve the $V^*(s')$ in Equation 3, we obtain a recursive definition of $Q(s,a)$.

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + \gamma max_{a'} Q(s',a')] \tag{5}$$

This recursive definition of Q forms the basis of the Q-Learning algorithm [17]. Q-learning starts with some initial values of $Q(s,a)$, and updates $Q(s,a)$ recursively using the actual reward received by the agent in a trial-and-error process. The complete learning process is depicted in the algorithm in Fig. 3.

In this algorithm, we assume that there is a single initial state, i.e. s_0, and a set of terminal states, i.e. S_r, in a given MDP. In the beginning, $Q(s,a)$ is initialized. For instance, $Q(s,a)$ can be set to 0 for all s and a. Then, the learning process is performed recursively. In each episode (round), the learner starts from the initial state s_0, and takes a sequence of actions by following the ϵ-greedy policy (which is introduced subsequently). The episode ends when the agent reaches a

Initialize $Q(s, a)$;
for each episode **do**
  $s \leftarrow s_0$;
  for $s \notin S_r$ **do**
    Choose $a \in A(s)$ based on ϵ-greedy policy;
    Execute a, observe reward r and new state s';
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$;
    $s \leftarrow s'$;
  end for
end for

Fig. 3. The Q-Learning Algorithm

terminal state $s \in S_r$. After executing each action, the learner updates $Q(s, a)$ using the following equation.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)] \tag{6}$$

On the right side of Equation 6, $r+\gamma max_{a'} Q(s', a')$ represents the newly observed reward (see Equation 5), and $Q(s, a)$ represents the previously observed reward. We can see that the function does not intend to use the newly observed value to completely replace the old $Q(s, a)$ value. Instead, it only updates a certain portion of the $Q(s, a)$ value, which is quantified by $\alpha(0 \leq \alpha \leq 1)$. α is called learning ratio, which is an important tuning factor in Q-Learning. Intuitively, the higher the learning ratio, the faster the learner will find the optimal policy. However, the higher the learning ratio, it is more likely that the learner be locked in a local optimal region.

The ϵ-greedy policy is used by the learner for executing the MDP during the learning. $\epsilon(\epsilon < 1)$ is the deterministic parameter of this policy. Given a state s, based on the current $Q(s, a)$ function, the ϵ-greedy policy chooses to execute the optimal action (i.e. $argmax_a Q(s, a)$) with a probability of $1 - \epsilon$. With a probability ϵ, the policy chooses a random action to execute. On the one hand, the ϵ-greedy policy ensures that $Q(s, a)$ is being optimized continuously. On the other hand, it guarantees that all the available actions are given chances to be tried out by the learner.

3.2 Applying Q-Learning to WSC-MDP

Our mechanism applies the Q-Learning algorithm to a WSC-MDP based service composition to determine the optimal or near-optimal policy at runtime. Rather than treating the Q-Learner as a learning system, our framework uses it directly as the execution engine of service compositions. Upon receiving a user request's, the system starts an execution of the service composition. It applies the ϵ-greedy policy of the Q-Learner to pick a service workflow to execute. This execution is in turn treated as an episode of the learning process. The ϵ-greedy policy and the Q-functions are updated afterwards, based on the newly observed reward.

By combining execution and learning, our framework achieves self-adaptivity automatically. When the environment changes, a service composition will change its policy accordingly, based on its new observation of reward. It does not require prior knowledge about the QoS attributes of the component services, but is able to achieve the optimal execution policy through learning.

Reward Assessment
To apply Q-Learning to a WSC-MDP, an important issue is to define the reward of the learning process. As the ultimate objective of our mechanism is to maximize user satisfaction. The reward should be a certain measure of user satisfaction. Information about user satisfaction can be obtained through two channels in a service composition system. This first channel is the QoS attributes that can be measured after the execution of each service. These QoS attributes include service fee, response time, availability, etc.. Our framework uses the following function to aggregate the various QoS attributes into a single reward value:

$$R(s) = \sum w_i \times \frac{Att_i^s - Att_i^{min}}{Att_i^{max} - Att_i^{min}} \tag{7}$$

where Att_i^s represent the observed value of the ith attribute of service s, and Att_i^{max} and Att_i^{min} represent the maximum and minimum values of Att_i for all services. w_i is the weighting factor of Att_i. w_i is positive if users prefer Att_i to be high (e.g. availability). w_i is negative if users prefer Att_i to be low (e.g. service fee and response time).

The second channel to assess user satisfaction is user feedback. A service composition system can allow a user to rate the composite service after each transaction. The rating is a kind of direct measure of the final reward received by the user. User feedback allow the learner to capture some properties that cannot be directly quantified. A typical property of such kind is user experience. Different from the QoS based reward, which can be measured at each learning step, users' feedback can only be measured at the final step of a episode. Fortunately, the Q-Learning approach is able to propagate the influence of the final reward to the intermediate services of the composition. Even when QoS attributes are not used, user feedbacks can still allow the learner to obtain a near-optimal policy.

Q-Function Initialization
Using Q-Learning, our framework does not need to know the QoS attributes of services to obtain the optimal execution policy. However, knowledge about QoS is still beneficial, as it allows the learner to obtain the optimal policy quickly. To incorporate the known QoS information into the learning process, we use this information to initialize the Q-Functions, i.e., $Q(s, a)$. In other words, for a service s whose QoS attributes are known, we calculate its initial $Q(s, a)$ by applying Equation 7 to its QoS attributes directly. For a service s whose QoS attributes are unknown, we approximate its initial QoS attributes by averaging the QoS values of other services, and apply Equation 7 to the approximated values to calculate $Q(s, a)$. As shown in our experiments, this initialization method is able to remarkably accelerate the learning process.

4 Experimental Evaluation

We conducted simulation to evaluate the properties of our service composition mechanism. This section presents some of the results.

4.1 Simulation Setup

We randomly generated WSC-MDP transition graphs to simulate service compositions. Each simulated WSC-MDP had a single initial state and two terminal states. The number of state nodes and the number of service nodes in a WSC-MDP graph ranged between 1,000 – 10,000 and 1,000 – 40,000 respectively. Each service node in a simulated WSC-MDP graph had at least one out-edge. The average number of out-edges per service node was set to 2.

We considered two QoS attributes of services. They were service fee and execution time. We assigned each service node in a simulated WSC-MDP graph with random QoS values. The values followed normal distribution. To simulate the dynamic environment, we periodically varied the QoS values of existing services based on a certain frequency.

We applied the Q-Learner introduced in Section 4 to execute the simulated service compositions. The reward function used by the learner were solely based on the two QoS attributes. We did not consider the reward information obtained from user feedback, as a simulation of user feedback may not reflect the reality. However, we believe that reward based on QoS can provide an adequate view of how a service composition can adapt to the environment through reinforcement learning. In the experiment results, without special announcement, the discount factor γ of the Q-Learner is set to 0.9, the learning rate α is set to 0.2 and the ϵ is set to 0.6.

Our experiments were performed on a 2.13GHz Intel Core2 PC with 2GB of RAM.

4.2 Efficiency of Learning

In the first stage of the evaluation, we studied how efficiently reinforcement learning would enable our service compositions to achieve the optimal execution policy. We assumed that the service composition system has zero knowledge about the QoS of the component services, and let the Q-Learner guide the service compositions to gradually reach the optimal policy. We conducted three sets of experiments to evaluate the efficiency.

In the first set of experiments, we fixed the number of states of the simulated WSC-MDP to 1,000 and the number of services for each state to 4. We varied the learning rate α from 0.1 to 0.6. For each α, we executed the service composition for 400 times. In other words, the learning was performed for 400 episodes for each α. We plot the cumulative reward of each episode in Fig. 4. As the figure shows, for all α, the cumulative rewards started converging before the 400th episodes. We can see that a higher learning rate can accelerate the learning process, whereas a smaller learning rate is helpful to avoid local optimality. As shown

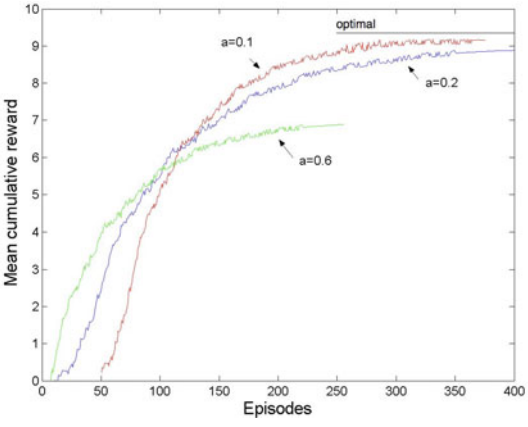


Fig. 4. Efficiency of Learning with Different Learning Rates

in Fig. 4, when α=0.6, although the cumulative reward increases very fast, the service composition failed to find the optimal policy. When α=0.1, the cumulative reward increases slowly, while it guarantee to achieve the optimal policy. When α=0.2, the learner seemed to achieve a good tradeoff between speed and effectiveness. Hence, we set α to 0.2 in the rest of our experiments. Fig. 4 also shows that the improvement made by the learning in the early stage is usually much higher than that in the late stage. This implies that a near-optimal policy can usually be identified much more quickly than the final optimal policy.

In the second set of experiments, we fixed the learning rate α to 0.2, but varied the parameter of the ϵ-greedy policy from 0.2 to 0.8. We studied how fast the cumulative reward converged to the optimal value during the learning. The cumulative reward is regarded to converge if the difference between its values in previous 10 consecutive episodes is below 1%. As shown in Fig. 5, the convergence time varies with ϵ. When $\epsilon = 0.6$, the convergence speed is the fastest. Hence, we set ϵ to 0.6 in the rest of our experiments.

Our third set of experiments aimed to analyze the relationship between the learning speed and the size of service composition.

Firstly, we fixed the number of services of each state to 4 and varied the number of states in a WSC-MDP graph from 1,000 to 10,000. We observed

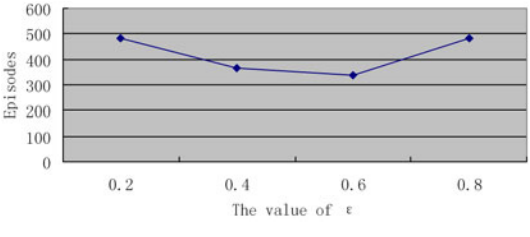
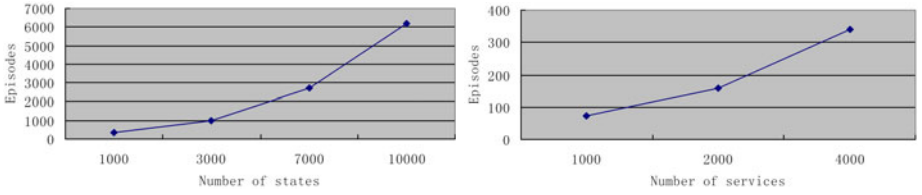


Fig. 5. Learning Speed vs ϵ



(a) Convergence Time vs # of States (b) Convergence Time vs # of Services

Fig. 6. Influence of Service Composition Size on Learning Speed

how fast the cumulative reward converged to optimal value during the learning. The results are shown in Fig. 6(a). As expected, the convergence time increases polynomially with the number of states. This is because the number of alternative workflows in a service composition usually increase exponentially with the number of states. Because of the efficiency of Q-Learning, the convergence time actually increases much slower than the number of workflows.

Secondly, we fixed the number of states to 1,000, and varied the number of alternative services in each state from 1 to 4. The results are shown in Fig. 6(b). The convergence time increases almost linearly with the number of services. This also shows the efficiency of Q-Learning, as the number of alternative workflows in a composition increases polynomially with the number of services.

A WSC-MDP graph with 10,000 states and 40,000 services seems to represent a fairly complex service composition. By using Q-Learning, our framework is able to identify the optimal execution policy with 6,000 episodes. In other words, once the composition has been executed for 6,000 times, it will reach the optimal status. As indicated by Fig. 4, a composition can reach a near-optimal status with a even faster speed.

4.3 Adaptivity to Changes

In the second stage of our evaluation, we studied how well our service compositions adapt to the changes of the environment.

In the first set of experiments, we simulated the changes of the environment by changing the QoS attributes of the services periodically. We assumed that the system have no knowledge about the services' QoS attributes, and let it rely on Q-Learning to learn the optimal execution policy. During the learning process, we changed the QoS attributes of the service in three kinds of frequencies. Namely, for every 100 episodes of learning, we varied 1%, 5% and 10% of the service's QoS attributes respectively. We observed how these changes would influence the effectiveness of learning.

Fig. 7 shows the growth of the cumulative reward during the learning process. We can see that by increasing the change rate we can delay a service composition to reach its optimal execution policy. However, the changes do not stop the optimization process. The execution policy is still being continually optimized when the learning goes on. When the turnover is as high as 10% per 100 episodes,

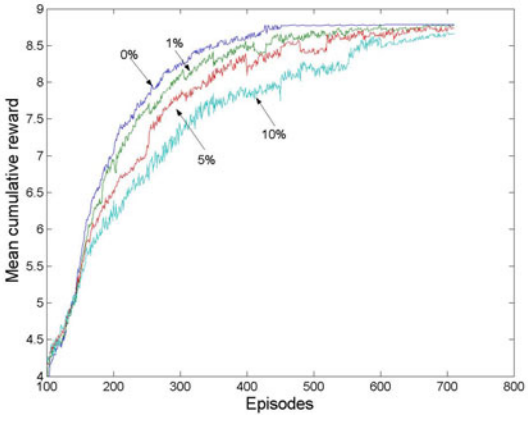


Fig. 7. Influence of Changes on Learning

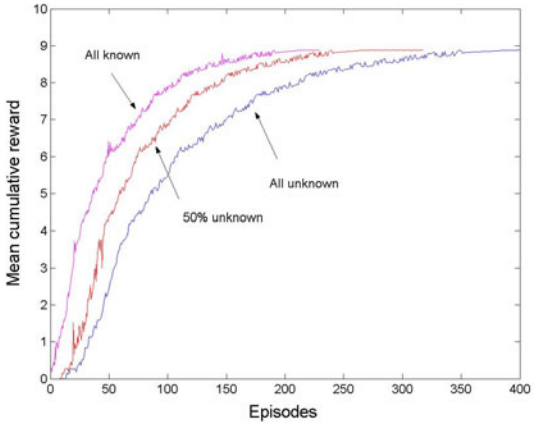


Fig. 8. Influence of Initial Knowledge

the service composition can still eventually reach a near optimal policy and stick to it afterwards.

In the final set of experiments, we studied how initial knowledge of services' QoS attributes can accelerate the learning process. We investigated in three cases. In the first case, all services' QoS attributes were known. In the second case, 50% of services' QoS attributes were known. In the last case, no QoS information is known. We applied the approach introduced in Section 4.2 to initialize the Q-function of the Q-Learner. The resulting learning processes are plotted in Fig. 8. The results show that the learning speed can be significantly improved by exploring the prior knowledge about services.

5 Related Work

The existing technologies for service composition mainly aim to achieve two objectives [19,1,18]. On the one hand, a service composition should provide the functionalities required by customers. On the other hand, the quality of a service composition should be optimized. Most of the previous approaches address the two issues in two separate phases.

In the first phase, a set of tools are used to create an abstract workflow of a service composition, which aims to satisfy users' requirements on functionality [13]. The applied techniques include AI planning [11,14], π-Calculus [16], Petri Nets [7], Model Checking [12], Finite State Machines [3], as well as Markov Decision Process (MDP) [5,6,4].

In the second phase, an optimal set of concrete services are selected to instantiate the abstract workflow. It generates a concrete service composition with the optimal Quality of Service (QoS). This phase is also known as service optimization. In [19], Zeng et al proposed a service quality model based on non-functional attributes, which has been widely used by the others [2,20,18] to perform service optimization. While the non-functional attributes of component services can reflect the quality of a service composition. However, they do not capture all the features of a composition. For example, user experience usually cannot be directly assessed using these attributes. In contrast, our learning based approach is able to take some implicit features into account. For instance, user feedbacks can be used by the learner to optimize the composition.

Recently, there have been a number of proposals for performing service optimization at runtime. An example is the work of Mei et al [9]. They applied social network analysis to rank services. Based on the ranking, their approach automatically selects the best service to run a workflow. Similarly, in [10], a tree-based algorithm was used to conduct runtime service selection. In theory, these two approaches are able to cope with evolving service quality. However, as they assume that information about QoS is known and up to date, they cannot deal with the real world cases where such information is unknown. Furthermore, they all assume a static workflow.

A fundamental difference between our approach and the previous approaches is that our service composition does not rely on an static workflow. By contrast, a service composition in our framework is able to integrate multiple alternative workflows and services. When executing the composition, our system decides which workflow and services to execute using its real-time knowledge. In addition, our system do not need to know the non-functional properties of services a-priori. Instead, it applies reinforcement learning to obtain the optimal solution by directly studying the results of execution. Therefore, our mechanism of service composition can be highly adaptive to a dynamic environment.

6 Conclusion

This paper introduces a novel framework for service composition. In contrast to most of the previous approaches, which develop service compositions upon

static workflow, our framework integrates multiple workflows and alternative services into a single composition. The concrete workflows and services used for execution are determined at runtime, based on the environment and the status of component services. By applying reinforcement learning, our framework is able to obtain the optimal execution policies of service compositions at runtime. Our experimental results show that our service compositions can obtain (near-)optimal execution policies efficiently, and they are highly adaptive to the changes of the component services.

References

1. Agarwal, V., Dasgupta, K., Karnik, N.M., Kumar, A., Kundu, A., Mittal, S., Srivastava, B.: A service creation environment based on end to end composition of web services. In: WWW, pp. 128–137 (2005)
2. Ardagna, D., Pernici, B.: Global and local qos guarantee in web service selection. In: Bussler, C.J., Haller, A. (eds.) BPM 2005. LNCS, vol. 3812, pp. 32–46. Springer, Heidelberg (2006)
3. Berardi, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Mecella, M.: Automatic service composition based on behavioral descriptions. Int. J. Cooperative Inf. Syst. 14(4), 333–376 (2005)
4. Chen, K., Xu, J., Reiff-Marganiec, S.: Markov-htn planning approach to enhance flexibility of automatic web service composition. In: ICWS, pp. 9–16 (2009)
5. Doshi, P., Goodwin, R., Akkiraju, R., Verma, K.: Dynamic workflow composition: Using markov decision processes. Int. J. Web Service Res. 2(1), 1–17 (2005)
6. Gao, A., Yang, D., Tang, S., Zhang, M.: Web service composition using markov decision processes. In: Fan, W., Wu, Z., Yang, J. (eds.) WAIM 2005. LNCS, vol. 3739, pp. 308–319. Springer, Heidelberg (2005)
7. Hamadi, R., Benatallah, B.: A petri net-based model for web service composition. In: ADC, pp. 191–200 (2003)
8. Kaelbling, L.P., Littman, M.L., Moore, A.P.: Reinforcement learning: A survey. J. Artif. Intell. Res (JAIR) 4, 237–285 (1996)
9. Mei, L., Chan, W.K., Tse, T.H.: An adaptive service selection approach to service composition. In: ICWS, pp. 70–77 (2008)
10. Oh, M., Baik, J., Kang, S., Choi, H.-J.: An efficient approach for qos aware service selection based on a tree-based algorithm. In: ACIS-ICIS, pp. 605–610 (2008)
11. Oh, S.-C., Lee, D., Kumara, S.R.T.: Effective web service composition in diverse and large-scale service networks. IEEE TSC 1(1), 15–32 (2008)
12. Rao, J., Küngas, P., Matskin, M.: Composition of semantic web services using linear logic theorem proving. Inf. Syst. 31(4-5), 340–360 (2006)
13. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
14. Shin, D.-H., Lee, K.-H., Suda, T.: Automated generation of composite web services based on functional semantics. J. Web Sem. 7(4), 332–343 (2009)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, Cambridge (1998)
16. Wang, Y.-L., Yu, X.-L.: Formalization and verification of automatic composition based on pi-calculus for semantic web service, December 1-30, vol. 1, pp. 103–106 (2009)

17. Watkins, C.J.C.H.: Learning from Delayed Rewards. PhD thesis, Kings College, Oxford (1989)
18. Yu, Q., Bouguettaya, A.: Framework for web service query algebra and optimization. TWEB 2(1) (2008)
19. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Trans. Softw. Eng. 30(5), 311–327 (2004)
20. Zeng, L., Ngu, A., Benatallah, B., Podorozhny, R., Lei, H.: Dynamic composition and optimization of web services. Distributed and Parallel Databases 24(1), 45–72 (2008)