Knowledge-Based Systems 62 (2014) 98–112

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Dynamic composition of Web services using efficient planners in large-scale service repository

Guobing Zou [a], Yanglan Gan [b,*], Yixin Chen [c], Bofeng Zhang [a]

[a] School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China
[b] School of Computer Science and Technology, Donghua University, Shanghai 201620, China
[c] Department of Computer Science and Engineering, Washington University in St. Louis, MO 63130, USA

ARTICLE INFO

Article history:
Received 7 July 2013
Received in revised form 16 February 2014
Accepted 1 March 2014
Available online 13 March 2014

Keywords:
Web service
Service composition
Automated planning
Deterministic planner
PDDL

ABSTRACT

Web services as independent software components are published by service providers over the Internet and invoked by service requesters for their desired functionalities. In many cases, however, there is no single service in a Web service repository satisfying a service request. So how to design an efficient method for composing a chain of connected services has become an important research issue. Recently, much research has been done into the search time reduction when finding a composite service. However, most methods take a long time for traversing all of the Web services in a service repository, thus it makes their response time significantly overrun a user's waiting patience. This paper develops an efficient approach for automatic composition of Web services using the state-of-the-art Artificial Intelligence (AI) planners, where a Web service composition (WSC) problem is regarded as a WSC planning problem. Unlike most traditional WSC methods that traverse a Web service repository many times, our approach converts a Web service repository into a planning domain in PDDL just once, which will only be regenerated when the Web service repository changes. This treatment substantially reduces the response time and improves the scalability of solving WSC problems. We have implemented a prototype system and conducted extensive experiments on large-scale Web service repositories. The experimental results demonstrate that our proposed approach outperforms the state-of-the-art.

1. Introduction

Web services are loosely coupled, self-descriptive, modular and Web-accessible distributed software components. They can be published in a Web service repository, discovered by software agents and composed as new value-added Web services. In most cases, the standard Web Service Description Language (WSDL[1]) is used to describe the input and output interface of a Web service for its functionality at the syntactical level. Meanwhile, Simple Object Access Protocol (SOAP) is commonly used for transferring messages and communications among Web services. Recently, Web service has become more and more important as it offers an extremely versatile and powerful tool to dynamically create distributed applications on demand. Its applications increase rapidly in many fields, such as electronic commerce, enterprise application integration and geographic information systems.

Web service discovery (WSD) aims at finding a service to fulfill a given service request. In many cases, however, no single service in a Web service repository is capable of fulfilling a service request. Therefore, it is mandatory to find a chain of services. They can be functionally connected together as a new composite service to satisfy the given service request. The problem of finding a group of connected and composable services from a Web service repository is called Web service composition (WSC). There are two ways in solving a WSC problem. The first way is to build a workflow business model [1] by domain experts with the help of GUI-based modeling software. However, it is impractical and error-prone when a Web service repository involves a large number of services. The other is to automatically and efficiently compose existing services from a large-scale Web service repository, which has become an important research issue in Web service community.

To efficiently solve the problem of dynamic composition of Web services, the main idea of this paper is to consider a WSC problem as a WSC planning problem, and then to use the state-of-the-art AI planners (e.g., Metric-FF [2] and SatPlan06 [3,4]) to find a composition

* Corresponding author. Address: School of Computer Science and Technology, 2999 North Renmin Road, Shanghai, China. Tel.: +86 21 67792291; fax: +86 21 67792106.

E-mail addresses: gbzou@shu.edu.cn (G. Zou), ylgan@dhu.edu.cn (Y. Gan), chen@cse.wustl.edu (Y. Chen), bfzhang@shu.edu.cn (B. Zhang).

[1] http://www.w3.org/TR/wsdl.

http://dx.doi.org/10.1016/j.knosys.2014.03.002
0950-7051/© 2014 Elsevier B.V. All rights reserved.

plan for a composition request. More specifically, we first convert all of the available services in a large-scale Web service repository into a planning domain in Planning Domain Definition Language (PDDL[2]), and then translate a composition request into a planning problem in PDDL. Finally, a WSC planning problem (consisting of a PDDL domain and a PDDL problem) is fed into an efficient AI planner, which can automatically find a composition plan for the given composition request.

The most distinguishing characteristic of our method lies in its shorter response time compared to other AI planning based WSC methods [5–9]. From the perspective of applicability, the response time is crucial to a WSC problem. It determines whether a WSC method can quickly respond to a composition request within a short period of time. Our proposed method only needs to traverse a Web service repository once. By doing so, time spent on parsing the Web service repository, which is a major part of the response time, can be greatly saved in our method. On the contrary, despite the fact that other AI planning based WSC methods, such as search by heuristic function [7,8] and planning graph model [9], have taken many efforts on reducing search time when finding a composite service, they repeatedly traverse all services in a Web service repository whenever a user submits a composition request.

The second characteristic of our proposed method is its compatibility with the exploitation of multiple automatic planners for service requesters. As it can be used with classical planners that support PDDL, users can choose from available desired planners according to their personalized requirements. For instance, SatPlan06 planner [3,4] can be used to find a composition plan with the minimum number of parallel steps, while Metric-FF planner [2] can find a feasible composition plan more quickly. Conversely, other AI planning based WSC methods, such as [5–9], take fixed composition modes or search strategies that cannot be served for complex user requirements in terms of their personalized preferences.

The third characteristic of our WSC method is its powerful applicability in real-world applications. Although most of recent WSC approaches [10–14] have taken into account semantic Web services described either in DAML-S[3] or OWL-S,[4] both providers and requesters have to describe the services in terms of ontological concepts to avoid semantic heterogeneity. The requesters may have difficulty in framing a service request correctly because of strict semantic rules to specify service functionality. Moreover, the construction of domain ontology for each area is also a challenging task with the help of domain experts. Currently, the dynamic composition of Web services in semantic level is still hard in practice from the view of the semantic annotation by service providers and composition request specification by service requesters. In comparison with the existing planner-based semantic WSC methods, we investigate WSC approach with Web services described in WSDL. It can be technically supported at present by industrial community. However, as more domain ontologies are being constructed and applied in real-world applications, semantic composition of Web services has emerged as a mainstream research direction in WSC community.

As described above, the innovation of this work can be summarized in two aspects. First, we propose a novel approach that translates a WSC problem to a WSC planning problem. It is solved by advanced AI planning techniques that can effectively compose Web services with shorter response time. Second, our WSC approach is based on standard PDDL specifications. It provides those service requesters with the flexibility of utilizing multiple efficient automated planners for personalized composition requests. However, Despite these advantages, the drawback of our approach is its syntactical matchmaking between preconditions and effects among WSC actions. That is, we currently mainly focus on in purely syntactic way rather than semantic composition of Web services. Therefore, this kind of matching scheme could mismatch the scenarios in real world applications, where WSC actions might be highly matched in terms of semantic way by similarity computation with a given threshold. In this aspect, although the proposed approach can solve large instances in a few seconds and outperform the existing WSC approach with faster response time and better scalability, to improve the correctness of WSC, our future work plans to consider semantic similarity calculation between the effects and preconditions of two WSC actions using semantic Web services posted on the website of ICEBE09,[5] where the input and output parameters are described and annotated by the taxonomy of concepts in OWL.

The proposed WSC approach has been implemented as a prototype system. Extensive experiments have been conducted on 18 groups of large-scale Web service repositories involving 81,464 Web services. The experimental results demonstrate that our proposed WSC approach using deterministic planning can significantly outperform WSPR [7,8], which is one of the state-of-the-art Web service composition methods.

The rest of this paper is organized as follows. Section 2 reviews the related work of WSC. Section 3 presents a motivating example in a real-world application. Section 4 formulates the WSC problem and WSC planning problem. Section 5 presents our approach for dynamic composition of Web services using planning. Section 5.1 illustrates its mapping mechanism in the translation process. Algorithms in Sections 5.2 and 5.3 are designed to generate a planning domain and a planning problem. Section 5.4 analyzes computational complexity of WSC planning problem translation. Section 5.5 finds a composition plan. Section 6 presents the WSC system architecture. Section 7 shows and analyzes extensive experimental results. Section 8 concludes the paper and discusses the future work.

2. Related work

According to the applied theories and techniques [15,16], WSC methods can be classified as workflow-based methods, AI planning based methods, graph theory based methods, and program synthesis based methods. In addition, there are other approaches used to address WSC problem, including algebraic process, π-calculus, petri net, model checking and finite-state machine [17].

In the workflow based methods, they first build an abstract business process model that consists of a set of tasks, control and data flow [15]. Then, each task in the process model contains a query clause used to search a real atomic service from a Web service repository. The authors in [18] presented an aggregated reliability (AR) model to measure the probability that the given state will lead to successful execution in the context, where each service may execute with some failure probability. Based on AR computation, it can dynamically select Web services performed on a composite service with more reliable execution. However, an abstract business workflow for a predetermined composite service needs to be modeled before dynamic service selection. The authors in [1] proposed a scientific workflow based editor, which allows scientists to query and compose distributed data sources. As a result, this kind of WSC methods are based on a workflow model and belong to a static composition approach. Moreover, it needs to be manually deployed by domain experts and GUI-based modeling softwares. Therefore, its dynamicity and flexibility are obviously reduced so that it is inappropriate for dynamic composition of Web services in a large-scale Web service repository.

[2] http://cs-www.cs.yale.edu/homes/dvm/.
[3] http://www.daml.org/services/.
[4] http://www.w3.org/Submission/OWL-S/.
[5] http://ws-challenge.georgetown.edu/wsc09/.

```
<message name="LocateMapWeather_Request">
    <part name="MSISDN" type="xsd:string"/>
    <part name="diameter" type="xsd:string"/>
</message>
<message name="LocateMapWeather_Response">
    <part name="map" type="xsd:string"/>
    <part name="weather" type="xsd:string"/>
</message>
```

(a) LocateMapWeather

```
<message name="LocatePhone_Request">
    <part name="MSISDN" type="xsd:string"/>
</message>
<message name="LocatePhone_Response">
    <part name="state" type="xsd:string"/>
    <part name="city" type="xsd:string"/>
    <part name="districtNum" type="xsd:string"/>
</message>
```

(b) LocatePhone

```
<message name="GetPosition_Request">
    <part name="city" type="xsd:string"/>
    <part name="districtNum" type="xsd:string"/>
</message>
<message name="GetPosition_Response">
    <part name="longitude" type="xsd:string"/>
    <part name="latitude" type="xsd:string"/>
</message>
```

(c) GetPosition

```
<message name="GetLatLon_Request">
    <part name="state" type="xsd:string"/>
    <part name="city" type="xsd:string"/>
</message>
<message name="GetLatLon_Response">
    <part name="longitude" type="xsd:string"/>
    <part name="latitude" type="xsd:string"/>
</message>
```

(d) GetLatLon

```
<message name="GetMap_Request">
    <part name="longitude" type="xsd:string"/>
    <part name="latitude" type="xsd:string"/>
    <part name="diameter" type="xsd:string"/>
</message>
<message name="GetMap_Response">
    <part name="map" type="xsd:string"/>
</message>
```

(e) GetMap

```
<message name="GetWeather_Request">
    <part name="state" type="xsd:string"/>
    <part name="city" type="xsd:string"/>
</message>
<message name="GetWeather_Response">
    <part name="weather" type="xsd:string"/>
</message>
```

(f) GetWeather

Fig. 1. The specifications of Web services map and weather in WSDL.

Many research efforts using AI planning techniques have been reported in recent years. Web service planner (WSPR) [7,8] presented an AI planning based algorithm to implement automatic composition of Web services. To find a feasible composition solution, it goes through two phases including forward search and regression search. During its search for a composition solution, a heuristic function is used to choose a service with the biggest contribution to match a subgoal. However, WSPR needs to parse all of the services in a service repository, whenever a composition request is submitted. Moreover, WSPR takes a locally optimal strategy in its regression search, which does not guarantee an optimal composition solution with the minimum number of Web services.

The authors in [9] proposed a service composition algorithm by planning graph model. The process of finding a composition solution is the construction of a planning graph. Although it is a complete algorithm (i.e., it guarantees to find a composition plan if one exists), when a new planning graph level is expanded by a set of applicable services, it just selects a subset of services in order, until they cover all of the output parameters of the candidate services. This possibly incurs redundant services in a feasible composition solution. A generic framework for service composition was presented in [14], where a service repository described by OWL-S process model is translated into an AI planning domain in PDDL. In addition, other WSC methods based on AI planning techniques focused on certain aspects, such as a semantic type matching algorithm considering those ambiguous state descriptions and incomplete operator definitions [19], WSC problems with the conformant form where services are partially controllable and observable [20], and WSC problems by AI planning at the knowledge level [21].

Besides the above planning based WSC methods, early AI planners have been applied to address a WSC problem. SHOP2 is a Hierarchical Task Network (HTN) planning system, which has been exploited for automatic Web service composition in [13]. All of the available services are first translated into a SHOP2 domain, and then SHOP2 planner recursively divides a composition task into many subtasks, until every subtask can be executed by a single service. Consequently, a composition plan can be generated and returned to users. Since SHOP2 differs from classical AI planners in its special expressions for planning domain and problem specifications, it is a WSC method with high dependency on the fixed AI planner and predefined subtasks partition. Hoffmann et al. [22,23] presented a planning-based method to formalize a special case WSC problem called “Strictly Forward Effects”. It takes integrity constraints (i.e., a set of axioms) as background theory specified by ontology to describe domain constraints between objects and their properties. Based on integrity constrains, a WSC problem is converted into a conformant planning problem under uncertainty with all of the possible initial states. An initial experiment has been conducted by Conformant-FF planner [24]. However, it is extensively hard to express input and output parameters and domain relations by predicates defined in an ontology, especially in a large-scale service repository. Furthermore, solving a conformant planning problem is much harder than that of a classical planning due to its belief updates.

In the graph theory based methods [5,6], Web services and their relationships are represented by a relational graph. The process of finding a composition solution is transformed to traverse relational graph and seek an accessible path, which starts from initial inputs and arrives at the desired outputs. In spite of facile implementation of this kind of WSC methods, when there are a huge number of services in a Web service repository, the relationships among services become so complicated that it needs to spend expensive cost constructing a relational graph. Furthermore, search space reduction in a complex relational graph also becomes a difficult task to rapidly find a service composition path.

In the program synthesis based WSC methods, the main idea [15] consists of three steps. Given a WSC problem, all of the Web services are firstly translated into logical axioms, and a composition request gets translated into a logical expression. Then, a theorem prover is taken to identify whether the logical expression can be proven by the logical axioms. Finally, a composition solution can be extracted from previous proof, or there is no solution to the original WSC problem.

3. Motivating example

This work was motivated by real-world applications. Fig. 1 shows an example, which includes six Web services on map and weather, and each of them is described in WSDL. The detailed specifications of these services are as follows.

(a) Given an MSISDN (one kind of telephone number) and a map diameter, Web service *LocateMapWeather* (Fig. 1(a)) returns a map with the specified diameter, as well as the current weather forecast of the city with regard to the given MSISDN.
(b) Given an MSISDN, Web service *LocatePhone* (Fig. 1(b)) responds to a state name, city name and district number of the MSISDN.
(c) Given a city name and a district number, Web service *GetPosition* (Fig. 1(c)) responds to a longitude and latitude of the city.
(d) Given a state name and a city name, *GetLatLon* (Fig. 1(d)) also returns a longitude and latitude of the city.
(e) Given a longitude, a latitude, and a map diameter, *GetMap* (Fig. 1(e)) responds to a map with the diameter.
(f) Given a state and a city name, *GetWeather* (Fig. 1(f)) returns the current weather forecast of the city.

Suppose that, a user provides an MSISDN (e.g., 314-629-2703) and a diameter (e.g., 15 miles) as the initial conditions. The user desires for a map of the city where the MSISDN locates with the given diameter scope, and the current weather forecast of that city. We consider two different Web service repositories as follows.

(1) Repository *A*: it consists of all of the six Web services: *A* = {*LocateMapWeather*, *LocatePhone*, *GetPosition*, *GetLatLon*, *GetMap*, *GetWeather*}.
(2) Repository *B*: it is comprised of the last five Web services: *B* = {*LocatePhone*, *GetPosition*, *GetLatLon*, *GetMap*, *GetWeather*}.

In repository *A*, *LocateMapWeather* can be directly applied, because its required parameters, MSISDN (314-629-2703) and diameter (15 miles), are given by the user. After its invocation and execution, a map (of St. Louis city) with 15 miles diameter scope and the current weather situation of St. Louis (e.g., 81 °F, partly cloudy) are both returned to the user.

In repository *B*, however, there is no single Web service that can fulfill the user's service request directly. Nevertheless, we can still find a chain of Web services composed together as a whole to satisfy the service request. One feasible solution to the given service request is as follows.

- *In the first step*, given an MSISDN (314-629-2703), we invoke *LocatePhone* to get a state name (Missouri), city name (St. Louis) and district number (314), respectively.
- *In the second step*, according to the returned state name (Missouri) and city name (St. Louis), we invoke *GetLatLon* to get the longitude ($90^\circ 12'$) and latitude ($38^\circ 37'$) of the city (St. Louis).
- *In the third step*, we invoke *GetMap* using longitude ($90^\circ 12'$), latitude ($38^\circ 37'$) and the given diameter (15 miles). After the invocation and execution, it returns a map (of St. Louis) with the diameter scope.
- *In the last step*, after receiving the state name (Missouri) and city name (St. Louis), we invoke service *GetWeather* to return the current weather forecast (81 °F, partly cloudy) of the city (St. Louis).

After the above four steps, the user gets a map (of St. Louis) within a specified diameter scope (15 miles). Furthermore, the current weather situation (81 °F, partly cloudy) is simultaneously returned to the user.

In this example, we can easily find a composite service to satisfy the service request by our manual deployment. However, as a large number of services can be available in a Web service repository, it tends to be a labor-intensive and difficult task. Thus, designing an automatic and efficient WSC method is mandatory and desirable for service requesters.

4. Problem formulation

In this section, we first formulate the WSC problem by definitions, and then present the WSC planning problem from AI planning perspective.

4.1. WSC problem

Web services are commonly described by an abstract data model, which employs a set of operations as its function description, a set of input and output messages as its request and response data, and the binding information as its invocation protocol [25]. In terms of a WSC problem, Web service is normally formalized by a two tuple with input and output interface parameter set. It is defined as below.

Definition 1 (*Web Service*[6]). A Web service, w, is a 2-tuple $\langle I, O\rangle$, where $I = \{I^1, I^2, \ldots\}$ is an interface parameter set, and each $I^i \in I$ is an input parameter. $O = \{O^1, O^2, \ldots\}$ is an interface parameter set, and each $O^i \in O$ is an output parameter. $w.I$ and $w.O$ are referred as I and O in w.

All input parameters of a Web service have to be given before it can be applied. After the invocation, all output parameters of a Web service are returned directly to users or for further use. In the motivating example, *LocatePhone* has an input interface parameter set *LocatePhone.I* = {*MSISDN*} and output interface parameter set *LocatePhone.O* = {*state*, *city*, *districtNum*}.

Definition 2 (*Web Service Repository*). A service repository, denoted as $W = \{w_1, w_2, \ldots\}$, is a set of available Web services, where each $w_i \in W$ is a service.

In the second scenario of the motivating example, the Web service repository *B* = {*LocatePhone*, *GetPosition*, *GetLatLon*, *GetMap*, *GetWeather*}.

In order to find a feasible composite service from a Web service repository, a service requester must provide a service composition request.

[6] Generally, a Web service consists of several operations, and each operation has an input and output interface parameter set. In order to simplify problem description, we denote one operation as one Web service here.

Definition 3 (*Composition Request*). A composition request, r, is a 2-tuple $\langle r_{in}, r_{out}\rangle$, where $r_{in} = \{r^1_{in}, r^2_{in}, \ldots\}$ is an interface parameter set provided as the initial inputs, and $r_{out} = \{r^1_{out}, r^2_{out}, \ldots\}$ is a goal specification with a set of desirable output parameters.

In the motivating example, The composition request of the user can be divided into an initial interface parameter set $r_{in} = \{MSISDN, diameter\}$ and a goal specification $r_{out} = \{map, weather\}$.

During the process of finding a composite service, a WSC method must check whether a service can be applicable by given an interface parameter set. Web services are composed by the following matching strategy.

Definition 4 (*Parameter Matching*). Given an interface parameter set $R = \{r^1, r^2, \ldots\}$, and a Web service $w = \langle I, O\rangle$. We define the following notations on parameter matching.

(1) If $w.I \subseteq R$, then all input parameters of w can be fully matched by R, i.e., w can be applicable to R. It is denoted as $R \succ w$.
(2) If $R \succ w$, then all the output parameters of w can be added to R after its being invoked. It is denoted as $R \oplus w = R \cup w.O$.

After applying w to $R, R \oplus w$ merges to be a new interface parameter set. In our example, we set R as initial interface parameter set $\{MSISDN, diameter\}$, and w as *LocatePhone*. Since the input interface parameter set of *LocatePhone* is $\{MSISDN\}$, it can be fully matched by the interface parameters $R = \{MSISDN, diameter\}$ (i.e., $w.I \subseteq R$), so *LocatePhone* can be applicable to R. After the invocation of *LocatePhone*, all of the output parameters of the service $\{state, city, districtNum\}$ are added to R, i.e., $R \oplus LocatePhone = R \cup LocatePhone.O = \{MSISDN, diameter\} \cup \{state, city, districtNum\} = \{MSISDN, diameter, state, city, districtNum\}$.

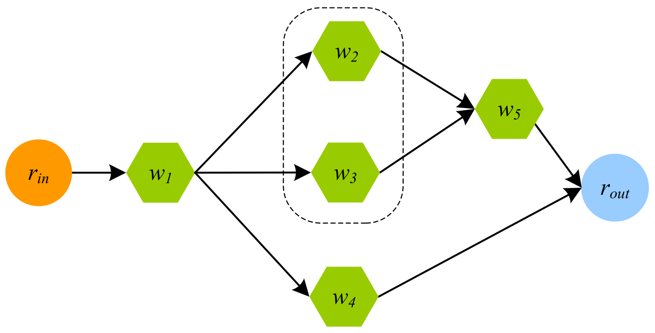
By using parameter matching scheme, we continuously apply services starting from an initial parameter set R and ending at a specified parameter set Q. It is defined as composition satisfiability below.

Definition 5 (*Composition Satisfiability*). Given two interface parameter set $R = \{r^1, r^2, \ldots\}$, $Q = \{q^1, q^2, \ldots\}$, and a set of Web services $\{w_1, w_2, \ldots, w_m\}$. If $(R \oplus w_i \oplus \cdots \oplus w_j) \supseteq Q$, $1 \leqslant i, j \leqslant m$, then $(w_i \otimes \cdots \otimes w_j) \propto (R \rightarrow Q)$ is denoted as composition satisfiability.

From Definition 5, we observe that after applying services $(w_i, \ldots, w_j)$ in order, all of the parameters in Q can be jointly matched by the parameters in R and all of the output parameters of services involved. Obviously, along with the invocation of Web services, the interface parameter set starting from R is monotonically growing, until it subsumes all of the parameters in Q. In the motivating example, we set its initial interface parameter set as $R = \{MSISDN, diameter\}$, and goal specification as $Q = \{map, weather\}$. For service repository B, $(R \oplus LocatePhone \oplus GetLatLon \oplus GetMap \oplus GetWeather) \supseteq Q$, so these four services can be applied from R to Q. We have the composition satisfiability $(LocatePhone \otimes GetLatLon \otimes GetMap \otimes GetWeather) \propto (R \rightarrow Q)$.

Definition 6 (*WSC Problem*). Given a composition request $r = \langle r_{in}, r_{out}\rangle$ and a service repository W, the WSC problem, denoted as $\langle r_{in}, r_{out}, W\rangle$, is to find an ordered sequence of services $O = (w_1, w_2, \ldots, w_n)$ from W $(O \subseteq W)$, such that $(w_1 \otimes w_2 \otimes \cdots \otimes w_n) \propto (r_{in} \rightarrow r_{out})$ is satisfiable.

Note that, given a WSC problem $\langle r_{in}, r_{out}, W\rangle$, a composition solution S to the WSC problem is an ordered sequence of services from W. More specifically, we start from r_{in}, and ends at an interface parameter set that subsumes all of the desired parameters in r_{out}, after applying the services involved in S in order. In such a case, more than one composition solution may exist for a WSC problem. For instance, we consider the WSC problem in the motivating example, its composable relationship between five Web services and the service composition request is illustrated in the following Fig. 2.



Legend: w_1*: LocatePhone* w_2*: GetPosition*
w_3*: GetLatLon* w_4*: GetWeather* w_5*: GetMap*

Fig. 2. The composable relationship between five services and composition request in the WSC problem, where $r_{in} = \{MSISDN, diameter\}$ and $r_{out} = \{map, weather\}$. Web services in the dashed area represent the disjunction relationship by providing the same output parameters $\{longitude, latitude\}$. Either w_2 or w_3 but not both of them is prerequisite by w_5.

In Fig. 2, $(w_1 \otimes w_2 \otimes w_5 \otimes w_4) \propto (r_{in} \rightarrow r_{out})$, $(w_1 \otimes w_4 \otimes w_3 \otimes w_5) \propto (r_{in} \rightarrow r_{out})$, and $(w_1 \otimes w_2 \otimes w_3 \otimes w_4 \otimes w_5) \propto (r_{in} \rightarrow r_{out})$ are satisfiable. Therefore, three of the composition solutions to the motivating example are $O_1 = (w_1, w_2, w_5, w_4)$, $O_2 = (w_1, w_4, w_3, w_5)$, and $O_3 = (w_1, w_2, w_3, w_4, w_5)$, respectively.

4.2. WSC planning problem

AI planning is an important technique with a variety of practical applications, such as controlling space vehicles, robots, and automated code synthesis and testing [26]. It starts from an initial state, then seeks a sequence of actions, and finally sets up an executable plan to arrive at a specified goal. So the motivation of AI planning is very practical: the need for information processing tools that provide affordable and efficient planning resources [26]. In particular, it can be applied to solve WSC problems. The reason is that they share high similarity between the process of finding a sequence of actions in an AI planning problem and composing a chain of Web services in a WSC problem.

To apply AI planning technique to solve a WSC problem, we formulate WSC planning problem as follows.

Definition 7 (*WSC State*). In a WSC problem setting, let L be a first-order language with finite predicates $P = \{p_1, p_2, \ldots\}$ and constant symbols $X = \{x_1, x_2, \ldots\}$. A WSC state comprises of a set of grounded propositions by P on X, denoted as $s = \{(p_1, x_1), (p_2, x_2), \ldots, (p_n, x_n)\}$, where $p_i \in P$ $(1 \leqslant i \leqslant n)$, and $x_i \in I \cup O$ $(1 \leqslant i \leqslant n)$ in a Web service w.

Predicates in L are defined to describe all possible planning states in a WSC problem, and constant symbols are from the parameters in a composition request, or input and output parameters of Web services. In this paper, we predefine a predicate $(yes\ ?x)$, which denotes the availability of an interface parameter x in a Web service or a composition request during WSC planning. In the motivating example, the user provides two initial parameters $\{MSISDN, diameter\}$. So the initial WSC state can be represented as $\{(yes\ MSISDN), (yes\ diameter)\}$.

All of the states in the powerset of P form a state space S, where a transition from one WSC state to another can be occurred by applying a WSC action.

Definition 8 (*WSC Action*). In a WSC problem setting, a WSC action is a triple $a = (name(a), precond(a), effects(a))$, where $name(a)$ is action name; $precond(a)$ is a set of propositions provided as its preconditions; $effects(a)$ is a set of positive propositions used as its effects.

A WSC action corresponds to a Web service in our WSC problem, its preconditions and effects are respectively generated from the input and output parameters of that service by using predefined predicate $(yes\ ?x)$. For example, *LocatePhone*, in the motivating example, is transformed into a WSC action with the same name. Therefore, it can be formalized as $a = (LocatePhone, \{(yes\ MSISDN)\}, \{(yes\ state), (yes\ city), (yes\ districtNum)\})$.

An action a can be applicable to a WSC state s, if and only if all of the preconditions of a are subsumed in s, i.e., $precond(a) \subseteq s$. After applying a to s, it is transmitted to a new WSC state by the following WSC state transition system.

Definition 9 (*WSC State Transition System*). In a WSC problem setting, let L be a first-order language. The WSC state transition system in L is a triple $\Sigma = (S, A, \gamma)$, where $S = \{s_1, s_2, \ldots\}$ is a state space represented by a set of WSC states; $A = \{a_1, a_2, \ldots\}$ is a set of WSC actions; $\gamma : S \times A \rightarrow S$ is a state transition function. For any action $a \in A$, if it can be applicable to a state $s \in S$, then $\gamma(s, a) = s \cup effects(a)$; otherwise $\gamma(s, a)$ is undefined.

In a WSC state transition system Σ, state space S has the closure feature under state transition function γ. In other words, $\forall s \in S$, after invoking an action $a \in A$ that can be applicable to s, γ transmits s to a new WSC state $s' = \gamma(s, a)$ such that $s' \in S$. For example, initial state s_0, in the motivating example, is expressed as $\{(yes\ MSISDN), (yes\ diameter)\}$. Since s_0 subsumes all of the preconditions of *LocatePhone*, i.e., $precond(LocatePhone) \subseteq s_0$, it can be applicable to state s_0. After applying *LocatePhone* to state s_0, state transition function γ transmits s_0 to a new state s_1, such that we have new state $s_1 = \gamma(s_0, LocatePhone) = s_0 \cup effects(LocatePhone) = \{(yes\ MSISDN), (yes\ diameter), (yes\ state), (yes\ city), (yes\ districtNum)\}$.

Based on the above definitions, we define the WSC planning problem as follows.

Definition 10 (*WSC Planning Problem*). In a WSC problem setting, let L be a first-order language with a set of finite predicates P and constant symbols X. The WSC planning problem, denoted as $\langle \Sigma, s_0, g \rangle$, is to find an ordered sequence of WSC actions $\pi = (a_1, a_2, \ldots, a_k)$, such that $g \subseteq \gamma(s_0, \pi)$.

(1) $\Sigma = (S, A, \gamma)$ is a WSC state transition system.
(2) s_0, the initial state, is an any WSC state in S.
(3) g, the goal specifications, involve a set of grounded propositions in P on X.

Given an ordered sequence of actions $\pi = (a_1, a_2, \ldots, a_k)$ and starting from s_0, we have $\gamma(s_0, \pi) = \gamma(s_0, (a_1, a_2, \ldots, a_k)) = \gamma(\gamma(s_0, a_1), (a_2, \ldots, a_k)) = \gamma(\ldots\gamma(\gamma(s_0, a_1), a_2)\ldots, a_k)$. That is, as we orderly apply actions in π, it leads to a series of state transitions $T = (s_0, s_1, \ldots, s_k)$, such that we have $s_1 = \gamma(s_0, a_1), \ldots, s_k = \gamma(s_{k-1}, a_k)$. Therefore, in a WSC planning problem, it starts from an initial state s_0, and ends at a WSC state s_k after applying actions $a_1, a_2, \ldots, a_k$, such that all of the goal specifications in g are subsumed in s_k, i.e., $g \subseteq s_k$. Particularly, although it is out of the scope of this paper, the ordered sequence of actions π could be converted into a composite service graph that may consist of multiple invocation relationships among services such as the inclusion of sequence, parallelism and choice.

From the above analysis, one observation is that there may exist a subset of state space, $S_g = \{s | (s \in S) \land (g \subseteq s)\}$, where each WSC state can satisfy all of the goal specifications in g. Thus, $\forall s \in S_g$, it contains all of the propositions in g. As a consequence, there are possibly multiple plans to a WSC planning problem. A composition plan is defined as follows.

Definition 11 (*Composition Plan*). In a WSC problem setting, let $\langle \Sigma, s_0, g \rangle$ be a WSC planning problem. A composition plan, $\pi = (a_1, \ldots, a_k)$, is an ordered sequence of actions, such that $g \subseteq \gamma(s_0, \pi)$.

From the above definition, a composition plan π must be a composition solution to its corresponding WSC problem, because an action uniquely maps to a Web service. Furthermore, in order to reduce communication cost during the execution of Web services, it is desired to find an optimal composition solution. There could be many optimization goals, e.g. depending on the cost of executing a service or quality aspects. In our problem setting, a minimum composition plan, π^*, is an ordered sequence of actions with as the least services involved as possible. Accordingly, a composition solution corresponding to π^* is also an optimal one to an original WSC problem, with the minimum number of Web services. For example, $O = (w_1, w_4, w_2, w_5)$ is an optimal composition solution in Fig. 2. However, $O' = (w_1, w_2, w_3, w_5, w_4)$ is a feasible composition solution instead of an optimal one, because w_2 and w_3 can both provide the same output parameters so that one of them is redundant within the composition solution O'.

Since a WSC planning problem is a classical AI planning problem, it can be represented by the classical description of AI planning. In classical planning, an AI planning problem can be divided into two parts: a planning domain and a planning problem, where planning domain denotes action specifications and planning problem involves initial conditions and goals. To generate a WSC planning problem $\langle \Sigma, s_0, g \rangle$, given a WSC problem $\langle r_{in}, r_{out}, W \rangle$, we translate its Web service repository W into a planning domain, and composition request $\langle r_{in}, r_{out} \rangle$ into a planning problem. The procedure of generating a WSC planning problem is elaborated in Section 5.

5. Automatic composition of Web services using planning

To solve a WSC problem, composing a chain of Web services can be converted into finding a sequence of actions in a WSC planning task. There are two steps for dynamic composition of Web services using automated planning technique. The first step is to take a WSC problem $\langle r_{in}, r_{out}, W \rangle$ as input, and translate it into a WSC planning problem. The second step is to apply the state-of-the-art automated planners in order to find a composition plan.

In the first step, we focus on the WSC planning problem translation. A service repository and a composition request are converted into a planning domain and a planning problem, respectively. Fig. 3 illustrates the translation process from a WSC problem to a WSC planning problem.

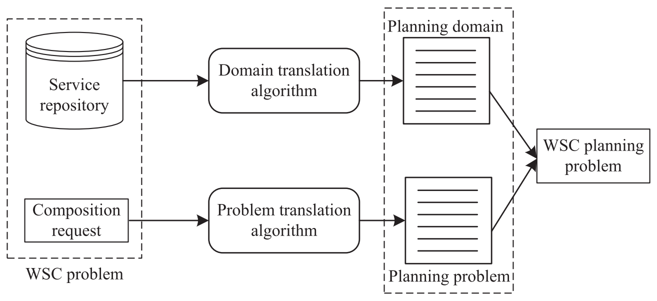


Fig. 3. Translation from a WSC problem to a WSC planning problem.

Given a WSC problem $\langle r_{in}, r_{out}, W\rangle$, the translation starts from the Web service repository W and the composition request $\langle r_{in}, r_{out}\rangle$. Then, the domain translation algorithm takes charge of parsing all of the services in W into a planning domain. The problem translation algorithm is responsible for translating $\langle r_{in}, r_{out}\rangle$ into a planning problem. Finally, the WSC planning problem consists of a planning domain and a planning problem. We elaborate the translation process in the following.

5.1. Mapping mechanism of WSC planning problem

The Planning Domain Definition Language (PDDL) [27] is an action-centered description language that is inspired by the well-known STRIPS formulations of AI planning problems. It is a standard encoding language for describing classical AI planning tasks, and has been widely adopted by most of the classical planners, such as Metric-FF [2] and SatPlan06 [3,4].

We use PDDL to describe a WSC planning problem, which is generated by two translation procedures in Fig. 3. (1) in planning domain translation, all of the available services in a service repository are modeled as actions in a PDDL domain. (2) in planning problem translation, initial parameters and goal specifications in a composition request are respectively modeled as an initial state and a goal specification in a PDDL problem. Fig. 4 illustrates the translation mechanism, which maps a Web service described in WSDL specification and a composition request to a PDDL domain and a PDDL problem.

As shown in Fig. 4, an operation in a Web service is mapped into an action in a PDDL domain. More specifically, inputs and outputs of a Web service are respectively modeled as preconditions and effects of its corresponding action. Meanwhile, all of the parameters from inputs and outputs of that service are added as problem objects in a PDDL problem. In a composition request, initial and goal parameters are respectively mapped to an initial state and a goal specification in the PDDL problem. The domain and problem translation algorithms are introduced in Sections 5.2 and 5.3.

5.2. WSC planning domain translation

Given a service repository $W = \{w_1, w_2, \ldots\}$, we present the algorithm designed to generate a planning domain in PDDL. Algorithm 1 describes its translation process.

Algorithm 1. Gen-WSC-Planning-Domain

```
Input: $W = \{w_1, w_2, \cdots\}$, Web service repository;
Output: $PD$, PDDL domain; $PO$, problem objects;
1  $PD(Types, Predicates, Actions) \leftarrow \emptyset$;
2  $Types \leftarrow \{string\}$; $Predicates \leftarrow \{(yes\ ?x)\}$;
3  $PO \leftarrow \emptyset$; $a \leftarrow \emptyset$;
4  foreach $w \in W$ do
5      $name(a) \leftarrow$ service name of $w$;
6      foreach $I^i \in w.I$ do
7          $precond(a) \leftarrow precond(a) \cup \{(yes\ I^i)\}$;
8      foreach $O^i \in w.O$ do
9          $effects(a) \leftarrow effects(a) \cup \{(yes\ O^i)\}$;
10     $Actions \leftarrow Actions \cup \{a\}$;
11     $PO \leftarrow PO \cup w.I \cup w.O$;
12     assign $a \leftarrow \emptyset$;
13 return $PD$, $PO$;
```

In terms of PDDL specifications, a planning domain contains three parts: types, predicates and actions. To simplify the problem formalization, in WSC planning translation we assume that all of the input and output parameters in a service have a uniform type *string*, and all of the grounded propositions in a WSC action by using finite predicates in P are used to denote the availability of an input or output parameter with a specified predicate $(yes\ ?p)$ in a WSC planning state. Although we only allow a simple data type and a predefined predicate for the presentation of service interface parameters, it can be extended to deal with the situations where there are complex data types and multiple predicates. The reason is that to find a composition plan for a deterministic WSC planning problem, most of the off-the-shelf efficient automatic planners (e.g., Metric-FF [2] and SatPlan06 [3,4]) apply heuristic search strategy for the syntactic matchmaking of effect and precondition propositions between two actions with logic reasoning techniques. It can accept multiple data types and first-order predicates. However, the condition is that, when service providers publish services in a repository, they need to specify input and output parameters with the provided data types and predicates. By doing so, our approach can automatically translate a set of disjoint Web services in a repository into a WSC planning domain without any manual deployment.

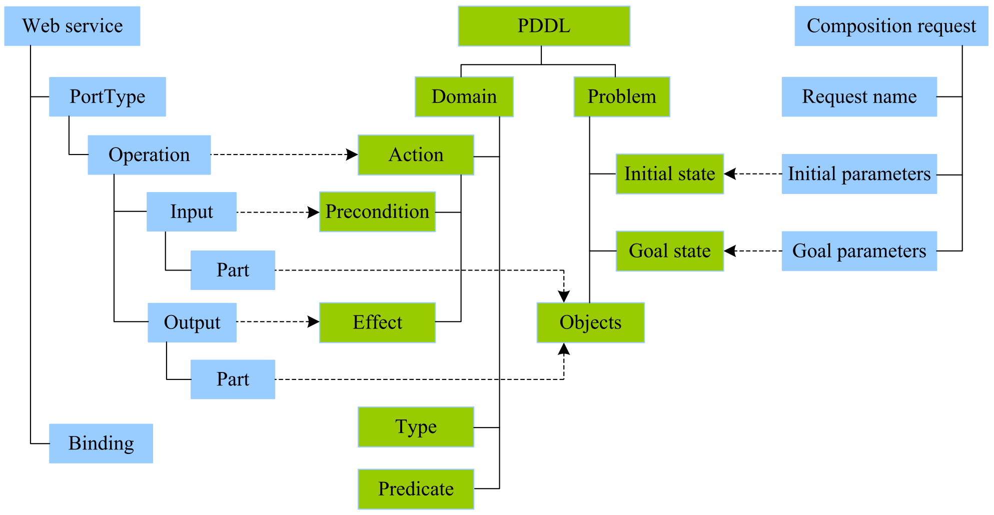


Fig. 4. Mapping mechanism from a Web service in WSDL specification and a composition request to a PDDL domain and a PDDL problem.

Based on this assumption, Algorithm 1 takes a service repository W with a set of Web services as an input, and the output is a WSC planning domain in PDDL with a set of parameters as problem objects. Each Web service $w \in W$ is translated into an action a in PDDL. That is, (1) service name of w is set as action name of a; (2) for each input parameter $I^i \in w.I$, we create a grounded proposition $(yes\ I^i)$, which is added to $precond(a)$ as one of the preconditions; (3) in the same way, for each output parameter $O^i \in w.O$, we use it to create a grounded proposition $(yes\ O^i)$, and add it to $effects(a)$ as one of the effects; (4) finally, all of the input and output parameters of Web service w are accumulated and put into PO as problem objects for WSC planning problem translation.

Taking the Web service repository B in Section 3 as an example, the five services, $\{LocatePhone, GetPosition, GetLatLon, GetMap, GetWeather\}$, are translated into their corresponding actions in a PDDL domain. Fig. 5 illustrates part of the generated planning domain. As shown, the planning domain involves predefined types $\{string\}$ and predicates $\{(yes\ ?x)\}$. Especially, the generated WSC planning domain involves five WSC actions and two of them in Fig. 5 correspond to Web services *LocatePhone* and *GetLatLon*, respectively.

In addition, during the WSC planning domain translation from Web services to WSC actions, we get a set of input and output parameters extracted from each service $w, PO = \{p | p \in (w.I \cup w.O), w \in W\}$. After the domain translation, $PO = \{MSISDN, state, city, districtNum, longitude, latitude, weather, diameter, map\}$. Each input or output parameter $p \in PO$ is used as a problem object in WSC planning problem translation shown in Algorithm 2.

5.3. WSC planning problem translation

Given a composition request $\langle r_{in}, r_{out} \rangle$, we devise the algorithm of generating a planning problem in PDDL. The translation process is shown in Algorithm 2.

The translation algorithm of generating a planning problem takes a composition request $\langle r_{in}, r_{out} \rangle$ and problem objects PO as its inputs. The output is a PDDL problem PP, which comprises of three parts: objects, initial state and goal specifications. We initially set each of them as $\emptyset$. Then, (1) for each parameter $p \in PO$, it is added as a problem object in PDDL problem; (2) for each initial parameter $r^i_{in} \in r_{in}$, we form a grounded proposition $(yes\ r^i_{in})$, which is put into the initial state as one of its request conditions. Thus, we have $Inits = \{(yes\ r^i_{in}) | r^i_{in} \in r_{in}\}$; (3) symmetrically, for each $r^j_{out} \in r_{out}$, a grounded proposition $(yes\ r^j_{out})$ is created and added to the goal specifications, i.e., $Goals = \{(yes\ r^j_{out}) | r^j_{out} \in r_{out}\}$.

Algorithm 2. Gen-WSC-Planning-Problem

```
Input: ⟨$r_{in}, r_{out}$⟩, composition request;
       $PO$, problem objects;
Output: $PP$, PDDL problem;
1 $PP(Objects, Inits, Goals) \leftarrow \emptyset$;
2 foreach $p \in PO$ do
3     $Objects \leftarrow Objects \cup \{p\}$;
4 foreach $r^i_{in} \in r_{in}$ do
5     $Inits \leftarrow Inits \cup \{(yes\ r^i_{in})\}$;
6 foreach $r^j_{out} \in r_{out}$ do
7     $Goals \leftarrow Goals \cup \{(yes\ r^j_{out})\}$;
8 return $PP$;
```

Taking the WSC problem $\langle r_{in}, r_{out}, W \rangle$ in Section 3 as an example, there are $r_{in} = \{MSISDN, diameter\}$ and $r_{out} = \{map, weather\}$. After the problem translation, part of the generated planning problem in PDDL is shown in Fig. 6. Each interface parameter in PO, generated in Algorithm 1, is used as a problem object. Initial state consists of two grounded propositions from r_{in}, $Inits = \{(yes\ MSISDN), (yes\ diameter)\}$. The goal specifications are translated from r_{out}, i.e., $Goals = \{(yes\ map), (yes\ weather)\}$.

5.4. Analysis of computational complexity

Let $\langle r_{in}, r_{out}, W \rangle$ be a WSC problem, where W is a Web service repository, r_{in} is a set of initial input parameters and r_{out} involves all of the parameters of goal specifications. For each $w \in W$, we denote the number of input and output parameters as $P^I_w = |w.I|$ and $P^O_w = |w.O|$, respectively. Suppose that $P = \max_{w \in W}\{|w.I| + |w.O|\}$ is used to denote the maximum number of input and output

```
(define (domain Map_Weather_Domain)
 (:requirements :typing)
 (:types string - object)
 (:predicates (yes ?x - string))

 (:action LocatePhone
   :precondition
    (and (yes MSISDN))
   :effect
    (and (yes state) (yes city) (yes districtNum)))

 (:action GetLatLon
   :precondition
    (and (yes state) (yes city))
   :effect
    (and (yes longitude) (yes latitude)))
 ...
)
```

Fig. 5. Part of the planning domain in PDDL for the Web service repository B in the motivating example.

```
(define (problem Map_Weather_Problem)
  (:domain Map_Weather_Domain)
  (:objects
   MSISDN - string
   state - string
   city - string
   districtNum - string
   longititude - string
   ...
  )
  (:init
   (yes MSISDN)
   (yes diameter)
  )
  (:goal (and
   (yes map)
   (yes weather)
  ))
)
```

Fig. 6. Part of the WSC planning problem given a service composition request $\langle r_{in}, r_{out} \rangle$ and the service repository B in the motivating example, where $r_{in} = \{MSISDN, diameter\}$, $r_{out} = \{map, weather\}$, and $B = \{LocatePhone, GetPosition, GetLatLon, GetMap, GetWeather\}$.

parameters among all of the services in W. We also assume that $P \ll |W|$ is satisfiable in a large-scale Web service repository.

The time computational complexity of generating a planning domain is determined by mapping Web services into actions as well as problem objects. We denote T_{PO} as the time complexity of generating problem objects. So the time complexity of planning domain translation can be calculated as: $T_{domain} = O(|W| + \sum_{w\in W}(|w.I| + |w.O|) + |W| + T_{PO} + |W|) = O(3 * |W| + \sum_{w\in W}(P^I_w + P^O_w) + T_{PO}) = O(3 * |W| + |W| * P + T_{PO})$. Furthermore, T_{PO} is dominated by the number of services involved in W and the number of parameters in each service w. More specifically, $\forall w \in W$, we identify whether each parameter $p \in (w.I \cup w.O)$ is subsumed in PO, i.e., $T_{PO} = O(\sum_{w\in W}(|w.I| + |w.O|)) = O(\sum_{w\in W}(P^I_w + P^O_w)) = O(|W| * P)$. Consequently, the complexity of domain translation is $T_{domain} = O(3 * |W| + |W| * P + |W| * P) = O((3 + 2P) * |W|)$. Since we have $P \ll |W|$ in a large-scale service repository, the time complexity of domain translation is $T_{domain} = O(|W|)$.

The time complexity of the planning problem translation for a WSC problem is dominated by three parts: the number of objects in PO, initial and goal parameter size in $\langle r_{in}, r_{out}\rangle$. Considering the worst case, without any repeated parameters exist among services. Thus, the time complexity of problem translation in PDDL is $T_{problem} = O(\sum_{w\in W}(|w.I| + |w.O|) + |r_{in}| + |r_{out}|) = O(\sum_{w\in W}(\left|P^I_w\right| + \left|P^O_w\right|) + |r_{in}| + |r_{out}|) = O(P * |W| + |r_{in}| + |r_{out}|)$. In terms of a large-scale service repository, since we have $P \ll |W|, |r_{in}| \ll |W|$, and $|r_{out}| \ll |W|$, the time complexity of problem translation is $T_{problem} = O(|W|)$.

From the calculation of time computational complexity, both WSC planning domain and WSC planning problem translation are linear algorithms in regard to the number of services in a Web service repository. Thus, they can be efficiently performed in a polynomial time.

5.5. Finding a composition plan

In the second step, we apply efficient automated planners to solve a WSC planning problem translated from a WSC problem in the first step. In order to verify the correctness of the WSC approach using planning, we solve the WSC problem described in the motivating example. Two highly efficient planners that support PDDL specifications, Metric-FF [2] and SatPlan06 [3,4], are applied to solve the WSC planning problem generated by planning domain and planning problem translation. Given the composition request in the motivating example, each of the two planners can find a composition plan.

(1) *Metric-FF planner.* The composition plan found by the Metric-FF contains an ordered sequence with four services, $\pi = (LocatePhone, GetPosition, GetMap, GetWeather)$.
(2) *SatPlan06 planner.* The found service composition plan by using the SatPlan06 also involves an ordered sequence with four Web services. However, its invocation steps differ from the order generated by the Metric-FF, $\pi' = (LocatePhone, GetWeather, GetPosition, GetMap)$.

From the verification of our WSC approach, Metric-FF and SatPlan06 can both find a composition plan that is directly mapped to a composition solution to the WSC problem in Section 3. We further prove that both planners can find a composition solution to the composition request in this scenario. There are three reasons. First, if there is a composition solution to a given WSC problem, $\langle r_{in}, r_{out}, W\rangle$, a composition plan must be found in its corresponding WSC planning problem, $\langle \Sigma, s_0, g\rangle$. Second, if a given WSC planning problem, $\langle \Sigma, s_0, g\rangle$, has a composition plan, AI planners can find it as long as they are implemented with complete planning algorithms. Third, since Metric-FF and SatPlan06 are both complete, our WSC system guarantees that it can find a composition solution to the given WSC problem in the motivating example. In this WSC motivating problem, both planners take less than 10 ms to find a composition plan. More performance analyses conducted on large-scale Web service repositories with 81,464 services are elaborated in the experimental evaluation.

5.6. Discussion

Although we only applied two efficient off-the-shelf automatic planners to find a composition plan for a WSC planning problem, there are also some recently developed state-of-the-art AI planners that have been taken into account for the dynamic composition of Web services and enterprise business process workflow integration in real-world applications. These approaches [10,28–33] are indeed used by corporations to create and manage operational business processes. Medical transport services are offered online by a variety of medical transport companies on the Internet. The service composition planner OWLS-XPlan [10,28] has been utilized in an agent based mobile eHealth system for emergency medical assistance (EMA) planning tasks, called Health-SCALLOPS. The planner runs on the server of a national EMA center to support the planning of patient relocation to selected hospitals, or patient repatriation. Furthermore, business processes coordinate the flow of activities within and between enterprises, where another important application area for automated planner is the creation of new processes in Business Process Management (BPM) at SAP corporation, one of the leading vendors of enterprise software. In this application [29,30], the model called Status and Action Management (SAM) was developed by SAP and used for planning to obtain a BPM planning application. SAM is compiled as a variant of planning language PDDL and an off-the-shelf fast forward (FF) planner is then adapted to help business experts create new processes simply by specifying the desired behavior in a real-time BPM process modeling environment, SAP NetWeaver platform. In addition, even though O-Plan [31,32] is an early automated planner, it takes an engineering approach to the construction of an efficient domain independent composition planning system which includes a mixture of AI and numerical techniques. It has been used in a wide variety of real applications [32,33], including air campaign planning, non-combatant evacuation operations, and biological pathway discovery. Finally, as a new designed classical AI planner LAMA [34] built on heuristic forward search, unlike the exploitation of binary state variables and multi-heuristic search to combine the landmark heuristic with a variant of the well-known FF heuristic [2], its core feature is the use of a pseudo-heuristic derived from landmarks, propositional formulas. Since LAMA builds on the fast downward planning system using finite-domain, it has potential application in dynamic composition of Web services with good performance of finding a composition plan.

Our proposed approach is original and distinguishes from existing methods with several advantages. Although many works have been done on dynamic composition of Web services using deterministic planning, our developed WSC planning system is superior to other existing AI planning WSC methods in its faster response time of finding a composition plan and better scalability on solving large-scale instances, thanks to the advanced deployment and exploitation of AI planning techniques. Additionally, we compile a WSC planning problem into a WSC planning problem in PDDL, so that our approach can be compatible with those off-the-shelf AI planners that supports the specifications with propositional logics and reasoning of planning domain and problem in PDDL. Furthermore, in contrast to other existing AI planning WSC methods, since it has powerful flexibility in the exploitation of different automatic planners, our approach can find a composition plan for service

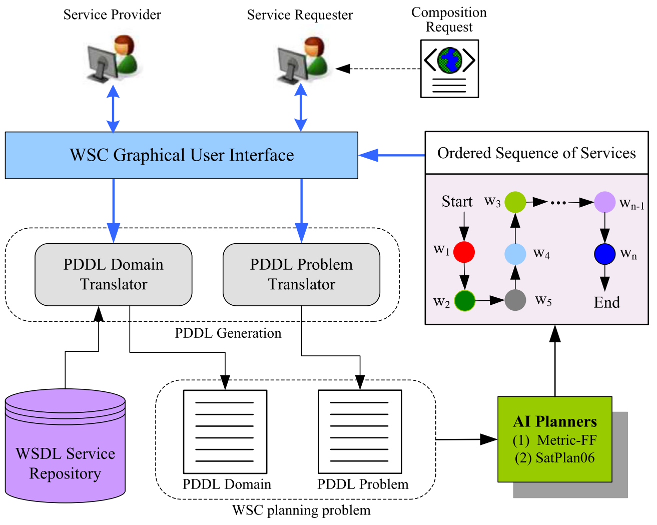


Fig. 7. The system architecture of Web service composition using efficient automated planners.

requesters with the consideration of their preferences, such as faster response time or the minimum parallel steps in the desired composition solution. The experimental results demonstrate that, our approach has significantly extended the capability of prior work by ensuring fast response time and good scalability when composing Web services in large-scale service repositories.

6. System architecture

To validate the feasibility and efficiency of our proposed WSC approach, we propose an architecture of Web service composition based on the state-of-the-art automated planners, designed for tests and experiments but not for production system. The system architecture is outlined in Fig. 7. It contains a WSC graphical user interface (GUI), a PDDL domain translator, a PDDL problem translator, a Web service repository with WSDL specification and two highly efficient AI planners. In addition, there are two kinds of participants: service provider and service requester. Service providers publish their services to the Web service repository for use. Service requesters consume those services offered by the service providers.

The process of dynamic composition of Web services based on automated planners is comprised of four steps. First, service providers publish their Web services by WSC graphical user interface to the WSC system, which stores all of the registered services into a Web service repository. Second, the WSC system reads all of the services from the Web service repository, and translates them into a planning domain by using the PDDL domain translator. Third, a service requester submits a composition request by the WSC graphical user interface, which is translated into a planning problem in PDDL by the WSC system using the PDDL problem translator. Based on the generated WSC planning problem (consisting of a PDDL domain and a PDDL problem), the WSC system invokes an AI planner (e.g., Metric-FF [2] or SatPlan06 [3,4]) to find a composition plan, which is mapped to a composition solution to the corresponding WSC problem. Finally, our WSC system returns the solution to the requester by the WSC graphical user interface.

7. Experimental evaluation

7.1. Experimental setup and datasets

In order to evaluate the effectiveness of our WSC method and compare its performance with the state-of-the-art WSC method, we developed a prototype system where all of the WSC components together with graphical user interface (GUI) in system architecture (Fig. 7) have been implemented by IDE Eclipse 3.5, Visual Editor 1.4 and Java. In addition, AI planners (Metric-FF and SatPlan06) applied in our experiments are integrated in the prototype. All of the experiments are performed on a PC with Intel Pentium(R) dual core processor 2.4 GHz and 1G RAM. Apart from AI planners tested on Ubuntu 10.04, other programs are run on Windows XP.

We have conducted extensive experiments on 81,464 Web services that are distributed in the 18 groups of large-scale Web service repositories. The datasets are published on ICEBE05[7] and can be freely downloaded from the website[8] of Web service challenge. These 18 groups of Web service repositories are categorized into Composition1 and Composition2, which are shown in Tables 1 and 2, respectively.

Table 1
Distributions of the tested Web service repositories in Composition1.

Composition1			
Dataset ID	Dataset	# of services	# of I/O
wsr-1-1	Composition1-20-4	2156	4–8
wsr-1-2	Composition1-20-16	2156	16–20
wsr-1-3	Composition1-20-32	2156	32–36
wsr-1-4	Composition1-50-4	2656	4–8
wsr-1-5	Composition1-50-16	2656	16–20
wsr-1-6	Composition1-50-32	2656	32–36
wsr-1-7	Composition1-100-4	4156	4–8
wsr-1-8	Composition1-100-16	4156	16–20
wsr-1-9	Composition1-100-32	4156	32–36

Table 2
Distributions of the tested Web service repositories in Composition2.

Composition2			
Dataset ID	Dataset	# of services	# of I/O
wsr-2-1	Composition2-20-4	3356	4–8
wsr-2-2	Composition2-20-16	6712	16–20
wsr-2-3	Composition2-20-32	3356	32–36
wsr-2-4	Composition2-50-4	5356	4–8
wsr-2-5	Composition2-50-16	5356	16–20
wsr-2-6	Composition2-50-32	5356	32–36
wsr-2-7	Composition2-100-4	8356	4–8
wsr-2-8	Composition2-100-16	8356	16–20
wsr-2-9	Composition2-100-32	8356	32–36

As shown in the service repository distributions, the number of services involved in a dataset ranges from 2156 to 8356, and the size of input or output parameters in a service ranges from 4–8, 16–20 to 32–36. In terms of the number of services and parameter size in a Web service repository, the easiest dataset to be dealt with is Composition1-20-4. On the contrary, the most difficult dataset is Composition2-100-32.

Each service repository either in the Composition1 or Composition2 has 11 composition requests for test. Accordingly, there are 11 composition solutions to their corresponding requests provided for verifying the correctness of WSC methods. Especially, no matter how differently composition requests have been used for the datasets in Composition1 or Composition2, their corresponding solutions involve the same number of services, as long as they have the same request ID and category.

From the distributions of Web services in Composition1 and Composition2 in Tables 1 and 2, note that since all of the input

[7] ICEBE05 provides a set of test data for both service composition and service discovery challenges.
[8] http://ws-challenge.georgetown.edu/ws-challenge/WSChallenge.htm.

and output parameters in a Web service have a uniform type, we predefine *string* as the data type for an interface parameter, and specify a predicate $(yes\ ?p)$ to represent the grounded propositions in preconditions and effects of a WSC action to denote the availability of the parameter p. As mentioned in Section 5.2, to deal with non-uniform data types and multiple predicates for WSC planning domain translation, we leverage the advanced automatic planners that apply heuristic search algorithm for the syntactic matchmaking of effect and precondition propositions between two actions with logic reasoning techniques, which can accept multiple data types and first-order predicates. Nevertheless, the condition is that service providers must specify input and output parameters with predefined complex data types and multiple predicates, when they publish Web services in a Web service repository.

7.2. Experimental results

The experimental results are shown in two ways. (1) the translation time spent on generating a planning domain and a planning problem on each dataset in Composition1 and Composition2. (2) the response time on all of the composition requests for 18 groups of datasets compared with WSPR [7,8].

7.2.1. Translation time for generating a planning domain and a planning problem

For each dataset in Composition1 and Composition2, translation time spent on generating its planning domain and planning problem is shown in Table 3. Especially, translation time for a planning problem generation is an average value on all of the 11 composition requests within a dataset. Thus, it is calculated by $\sum_{i=1}^{11} Translate(r_i)/11$, where $Translate(r_i)$ is the translation time for a composition request r_i, where $1 \leqslant i \leqslant 11$.

The results summarized in Table 3 indicate that the average time for generating a planning problem in PDDL can be taken within a short period of time. It ranges from 1.36 ms to 14.27 ms. In the meantime, with an increasing number of services and parameter size involved in different Web service repositories, the translation time taken for generating a planning domain rises with slow speedup. It ranges from 74,031 ms to 1,103,375 ms. In particular, although translation time lasts much longer for the generation of a planning domain than that for a planning problem, the task to translate a Web service repository can be performed offline just for one time, and it only needs to be partly recomputed when that Web service repository has changed (e.g., there are new services added or existing services have eliminated from the Web service repository).

7.2.2. Response time on finding a composition plan

From the view of practicability in real-world applications, the response time is of vital importance to a WSC method, because it determines whether a feasible composition solution can be rapidly returned to users within a short period of time. Therefore, we employ response time as the evaluation metric to compare the efficiency of our method with the state-of-the-art WSC method WSPR [7,8] throughout the experiment.

Definition 12 (*Response Time*). Given a Web service repository W, and its corresponding planning domain D. The response time of a WSC method m, denoted as $RT(m)$, holds the duration, when it starts from submitting a composition request r by a user and ends at receiving a composition solution or failing to find it. The response time in our method and WSPR are respectively defined as:

(1) $RT(Metric\text{-}FF) = Translate(r) + Parse(P) + Parse(D) + Plan(\pi)$;
(2) $RT(SatPlan06) = Translate(r) + Parse(P) + Parse(D) + Plan(\pi)$;
(3) $RT(WSPR) = Parse(r) + Parse(W) + Search(O)$;

The response time in our WSC method lasts the duration, including translating a composition request r to its corresponding planning problem P, parsing planning problem P and planning domain D, and applying an AI planner to find a composition plan π. On the other hand, WSPR takes its response time by parsing composition request r, parsing Web services in a service repository W, and searching a composition solution O by its forward and regression search strategy.

We denote WSPR as the WSC method [7,8], which is one of the state-of-the-art AI planners that can be directly used for Web service composition. To simplify the expression, Metric-FF is marked as our WSC method based on Metric-FF planner, and SatPlan06 stands for our WSC method by using SatPlan06 planner. We test the response time on 11 composition requests in each dataset within 18 groups of Web service repositories among three approaches, i.e., Metric-FF, SatPlan06, and WSPR. The 11 composition requests tested on each dataset are separate trials and could occur in any order. The results of response time are illustrated in Fig. 8.

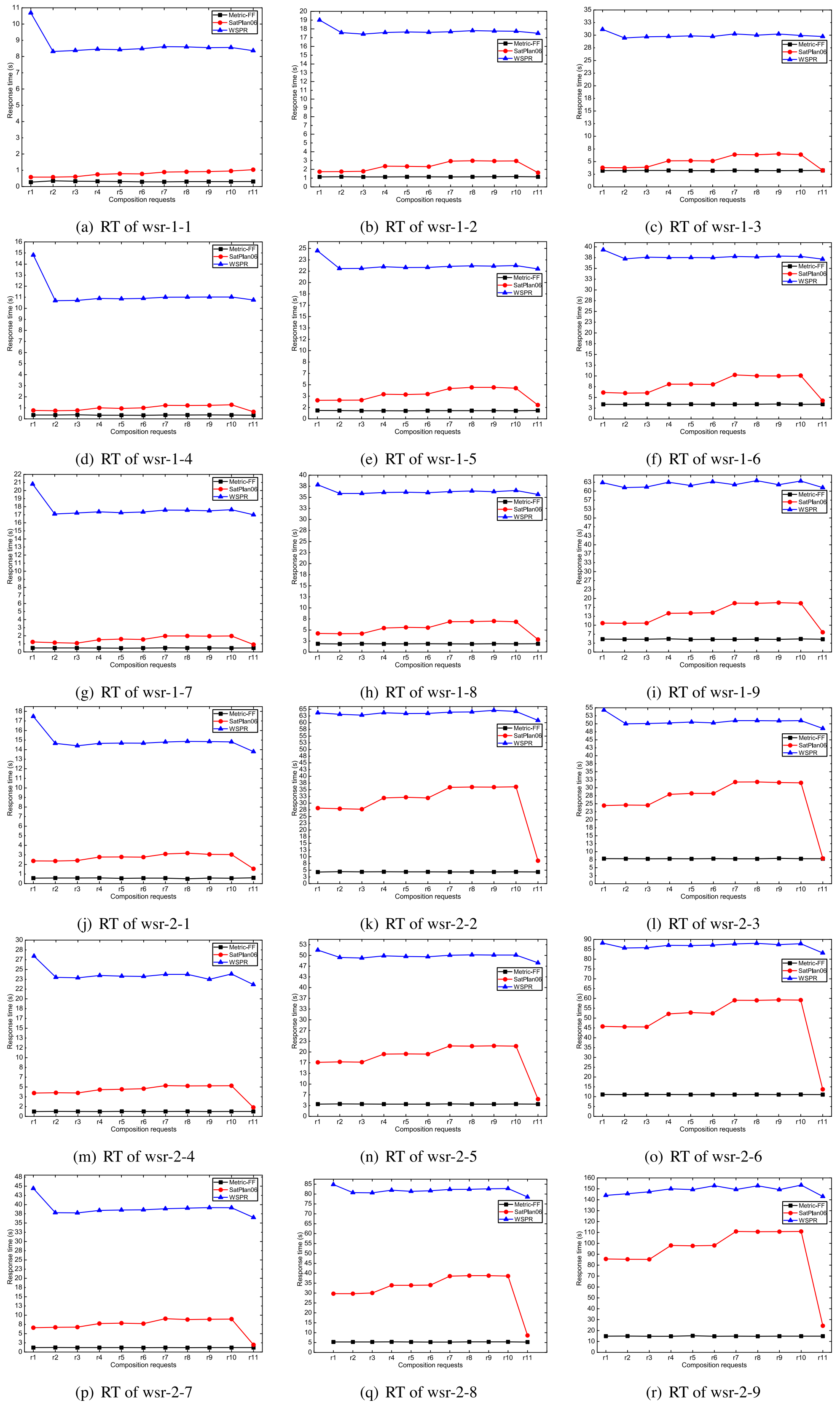
7.3. Experimental analysis

From the above experimental results, we summarize the comparisons and analyses between our WSC method using efficient planners and WSPR as follows.

(1) Our method based on AI planners (Metric-FF, SatPlan06) can find a composition solution faster than WSPR in response time. To be more precise, the response time of using Metric-FF ranges from 0.27 s to 14.915 s; the minimal response time of SatPlan06 is 0.58 s and the maximum is 110.95 s. Correspondingly, the response time of WSPR ranges from 8.313 s to 153.422 s. The main reason for the difference between our method and WSPR in response time is that

Table 3
Translation time taken for generating a planning domain and a planning problem on all of the 18 groups of datasets in Composition1 and Composition2. Column *Domain* represents the translation time of generating a planning domain in PDDL. Column *Problem* is the average time of translating 11 composition requests in a dataset to a planning problem in PDDL.

Composition1			Composition2		
Dataset ID	Domain (ms)	Problem (ms)	Dataset ID	Domain (ms)	Problem (ms)
wsr-1-1	74,031	1.36	wsr-2-1	164,922	1.36
wsr-1-2	75,641	1.45	wsr-2-2	625,453	2.82
wsr-1-3	77,203	2.82	wsr-2-3	175,735	8.55
wsr-1-4	109,765	1.45	wsr-2-4	400,140	2.91
wsr-1-5	112,250	2.91	wsr-2-5	412,312	2.82
wsr-1-6	112,953	2.82	wsr-2-6	417,438	11.18
wsr-1-7	250,922	1.45	wsr-2-7	964,031	2.82
wsr-1-8	253,000	2.91	wsr-2-8	972,469	1.45
wsr-1-9	260,172	14.27	wsr-2-9	1,103,375	8.45



(a) RT of wsr-1-1 (b) RT of wsr-1-2 (c) RT of wsr-1-3

(d) RT of wsr-1-4 (e) RT of wsr-1-5 (f) RT of wsr-1-6

(g) RT of wsr-1-7 (h) RT of wsr-1-8 (i) RT of wsr-1-9

(j) RT of wsr-2-1 (k) RT of wsr-2-2 (l) RT of wsr-2-3

(m) RT of wsr-2-4 (n) RT of wsr-2-5 (o) RT of wsr-2-6

(p) RT of wsr-2-7 (q) RT of wsr-2-8 (r) RT of wsr-2-9

Fig. 8. The response time of each 11 composition requests on their corresponding dataset in 18 groups of Web service repositories among three WSC approaches Metric-FF, SatPlan06 and WSPR.

the applied planners can more rapidly parse planning domain, while WSPR takes much longer to parse all of the services in a Web service repository. Also, the planning time taken by Metric-FF and SatPlan06 outperforms the search time in WSPR. Especially, our WSC method using Metric-FF can best satisfy all of the composition requests because its response time to the hardest dataset (Composition2-100-32) still remains within a short period of time.

(2) Our WSC method using Metric-FF is significantly faster than that SatPlan06 when a dataset becomes more difficult to handle. The major reason is that SatPlan06 needs to convert WSC planning problem into a constraint satisfaction problem (CSP), before it starts invoking a solver to find a composition plan. However, one benefit of SatPlan06 for service requesters is that a composition plan with the minimum number of parallel steps can be found if it exists from a service repository.

(3) The scalability of our WSC method is superior to WSPR. One fact is that all of the WSC methods take much longer time to find a composition solution given a more difficult Web service repository. However, along with the increase of the number of Web services and parameter size in a Web service repository, the response time taken by planners Metric-FF and SatPlan06 rises substantially slower than that taken by WSPR. Tables 4 and 5 show the absolute increase and its rate of response time from service number and parameter size among Metric-FF, SatPlan06 and WSPR.

With regard to the change of service number, Metric-FF has the lowest increase rate of average response time on all of the Web service repositories as well as the absolute increase of response time. In most cases, SatPlan06 has a worse increase rate of average response time than WSPR, but its absolute increase of response time is extensively slower than WSPR. Symmetrically, although WSPR has the lowest increase rate of average response time along with the change of parameter size, its absolute increase of response time is the quickest. In general, Metric-FF keeps in a stable and slow

Table 4
The absolute increase and its rate of response time among Metric-FF, SatPlan06 and WSPR along with the change of service number. $|P|$ gives the parameter size of input or output of a service. $|W|$ is the number of services in a repository. *ART* is the average response time on all of the 11 composition requests in a Web service repository. *RTI* is the absolute increase of average response time compared with the preceding *ART*. *Rate* is the increase rate of average response time.

$\|P\|$	$\|W\|$	*Metric – FF*			*SatPlan*06			*WSPR*		
		ART	*RTI*	*Rate*	*ART*	*RTI*	*Rate*	*ART*	*RTI*	*Rate*
4–8	2156	0.306	–	–	0.800	–	–	8.674	–	–
	2656	0.344	0.038	0.124	0.980	0.180	0.225	11.242	2.568	0.296
	4156	0.491	0.147	0.427	1.539	0.559	0.570	17.665	6.423	0.571
16–20	2156	1.149	–	–	2.334	–	–	17.753	–	–
	2656	1.198	0.049	0.043	3.565	1.231	0.527	22.478	4.725	0.266
	4156	1.873	0.675	0.563	5.419	1.854	0.520	36.278	13.800	0.614
32–36	2156	3.224	–	–	5.076	–	–	29.988	–	–
	2656	3.422	0.198	0.061	7.910	2.834	0.558	37.726	7.738	0.258
	4156	4.792	1.370	0.400	14.221	6.311	0.798	62.629	24.903	0.660
4–8	3356	0.569	–	–	2.672	–	–	14.878	–	–
	5356	0.845	0.276	0.485	4.391	1.719	0.643	24.046	9.168	0.616
	8356	1.214	0.279	0.437	7.373	2.982	0.679	38.934	14.888	0.619
16–20	5356	3.817	–	–	18.307	–	–	49.776	–	–
	6712	4.359	0.542	0.142	30.210	11.903	0.650	63.430	13.654	0.274
	8356	5.320	0.961	0.220	32.235	2.025	0.067	81.791	18.361	0.289
32–36	3356	7.819	–	–	26.609	–	–	50.719	–	–
	5356	11.098	3.279	0.419	49.489	22.880	0.860	86.794	36.075	0.711
	8356	14.833	3.735	0.337	92.536	43.047	0.870	148.807	62.013	0.714

Table 5
The absolute increase and its rate of response time among Metric-FF, SatPlan06 and WSPR along with the change of parameter size. Column notations have the same meaning with Table 4.

$\|W\|$	$\|P\|$	*Metric – FF*			*SatPlan*06			*WSPR*		
		ART	*RTI*	*Rate*	*ART*	*RTI*	*Rate*	*ART*	*RTI*	*Rate*
2156	4–8	0.306	–	–	0.800	–	–	8.674	–	–
	16–20	1.149	0.843	2.755	2.334	1.534	1.918	17.753	9.079	1.047
	32–36	3.224	2.075	1.806	5.076	2.742	1.175	29.988	12.235	0.689
2656	4–8	0.344	–	–	0.980	–	–	11.242	–	–
	16–20	1.198	0.854	2.483	3.565	2.585	2.638	22.478	11.236	0.999
	32–36	3.422	2.224	1.856	7.910	4.345	1.219	37.726	15.248	0.678
4156	4–8	0.491	–	–	1.539	–	–	17.665	–	–
	16–20	1.873	1.382	2.815	5.419	3.880	2.521	32.278	14.613	0.827
	32–36	4.792	2.919	1.558	14.221	8.802	1.624	62.629	30.351	0.940
3356	4–8	0.569	–	–	2.672	–	–	14.878	–	–
	32–36	7.819	7.250	12.742	26.210	23.538	8.809	50.719	35.841	2.409
5356	4–8	0.845	–	–	4.391	–	–	24.046	–	–
	16–20	3.817	2.972	3.517	18.307	13.916	3.169	49.776	25.730	1.070
	32–36	11.098	7.281	1.908	49.489	31.182	1.703	86.794	37.018	0.744
8356	4–8	1.214	–	–	7.373	–	–	38.934	–	–
	16–20	5.320	4.106	3.382	32.235	24.862	3.372	81.791	42.857	1.101
	32–36	14.833	9.513	1.788	92.536	60.301	1.871	148.807	67.016	0.819

increase of response time, as Web service repository becomes more complex (i.e., service number and parameter size). SatPlan06 is relatively more sensitive than Metric-FF in its response time with the increase of service number and parameter size. However, WSPR fluctuates with the quickest change in its response time.

(4) Our WSC method guarantees completeness in finding a composition solution. Since all of the applied planners (Metric-FF and SatPlan06) are implemented with complete algorithms, our WSC method can find a composition plan if one exists. The experimental results demonstrated that Metric-FF and SatPlan06 can both find a composition plan with no exceptions to all of the 11 composition requests on each dataset. The found composition plans can be exactly matched with the provided solutions that are optimal to all of the composition requests. Meanwhile, WSPR can also find corresponding composition solutions with the least number of services to all of the composition requests on each dataset.

Based on the experimental results and analyses, it comes to a conclusion that our proposed WSC method using Metric-FF planner outperforms SatPlan06 and WSPR both in its response time and scalability.

8. Conclusions and future work

The ability to automatically and efficiently compose Web services can potentially simplify the implementation of business processes. This paper presents an efficient approach for dynamic composition of Web services using the state-of-the-art automated planners. Unlike most existing WSC methods that need to parse all of the services in a Web service repository, whenever users submit a composition request, our WSC method translates a Web service repository into a planning domain in PDDL just once, and it needs to be recomputed only when the Web service repository has changed. Therefore, our WSC method can rapidly respond to a composition request. The extensive experiments conducted on large-scale Web service repositories indicate that our proposed WSC method using Metric-FF outperforms the state-of-the-art both in its response time and scalability.

There are several limitations that will be further addressed in our near future research, including the support of semantic Web service composition, replanning of composite service during the execution failure, and efficiently updating WSC planning domain. These efforts involve adding semantic similarity calculation to WSC planning problem and AI planners for more effective matching between input and output parameters of Web services, automatically updating planning domain in PDDL by monitoring the change of Web service repository, and verifying our WSC approach on more real-world Web service repositories with more abundant textual descriptions.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (61303096,61300100), Shanghai Natural Science Foundation (13ZR1454600,13ZR1451000), an Innovation Program of Shanghai Municipal Education Commission (14YZ017), a Specialized Research Fund for the Doctoral Program of Higher Education (20133108120029), a National Science Foundation (NSF) Grant IIS-0713109, and a Microsoft Research New Faculty Fellowship.

We thank Jörg Hoffmann, Henry Kautz and Bart Selman for their open sources of AI planners Metric-FF and SatPlan06. We would like to appreciate the two anonymous reviewers for their insightful suggestions and constructive comments.

References

[1] I. Altintas, E. Jaeger, K. Lin, et al., A Web service composition and deployment framework for scientific workflows, in: Proceedings of the IEEE International Conference on Web Services (ICWS), 2004.
[2] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. (JAIR) 14 (1) (2001) 253–302.
[3] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1999, pp. 318–325.
[4] H. Kautz, B. Selman, J. Hoffmann, SatPlan: planning as satisfiability, in: Abstracts of the International Planning Competition (IPC), 2006.
[5] R. Aydogan, H. Zirtiloglu, A graph-based Web service composition technique using ontological information, in: Proceedings of the IEEE International Conference on Web Services (ICWS), 2007, pp. 1154–1155.
[6] S. Hashemian, F. Mavaddat, A graph-based approach to Web services composition, in: Proceedings of the International Symposium on Applications and the Internet (SAINT), 2005.
[7] S. Oh, D. Lee, S. Kumara, Web service planner (WSPR): an effective and scalable Web service composition algorithm, Int. J. Web Services Res. (IJWSR) 4 (1) (2007) 1–22.
[8] S. Oh, D. Lee, S. Kumara, Effective Web service composition in diverse and large-scale service networks, IEEE Trans. Services Comput. (TSC) 1 (1) (2008) 15–32.
[9] X. Zheng, Y. Yan, An efficient syntactic Web service composition algorithm based on the planning graph model, in: Proceedings of the IEEE International Conference on Web Services (ICWS), 2008, pp. 691–699.
[10] M. Klusch, A. Gerber, M. Schmidt, semantic Web service composition planning with OWLS-XPlan, in: Proceedings of AAAI Fall Symposium on Semantic Web and Agents, 2005.
[11] J. Peer, A PDDL based tool for automatic Web service composition, in: Proceedings of the International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR), 2004, pp. 149–163.
[12] M. Sheshagiri, M. Desjardins, T. Finin, A planner for composing services described in DAML-S, in: Proceedings of the ICAPS Workshop on Planning for Web Services, 2003.
[13] E. Sirin, B. Parsia, D. Wu, et al., HTN planning for Web service composition using SHOP2, J. Web Semantics (JWS) 1 (4) (2004) 377–396.
[14] B. Yang, Z. Qin, Composing semantic Web services with PDDL, Inf. Technol. J. 9 (1) (2010) 48–54.
[15] J. Rao, X. Su, A survey of automated Web service composition methods, in: Proceedings of the International Workshop on Semantic Web Services and Web Process Composition (SWSWPC), 2005, pp. 43–54.
[16] B. Srivastava, J. Koehler, Web service composition-current solutions and open problems, in: Proceedings of the ICAPS Workshop on Planning for Web Services, 2003.
[17] N. Milanovic, M. Malek, Current solutions for Web service composition, IEEE Internet Comput. 8 (6) (2004) 51–59.
[18] S. Hwang, E. Lim, C. Lee, et al., Dynamic Web service selection for reliable Web service composition, IEEE Trans. Services Comput. (TSC) 1 (2) (2008) 104–116.
[19] M. Carman, L. Serafini, P. Traverso, Web service composition as planning, in: Procceedings of the ICAPS Workshop on Planning for Web Services, 2003.
[20] G. Giacomo, R. Masellis, F. Patrizi, Composition of partially observable services exporting their behaviour, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2009.
[21] M. Pistore, A. Marconi, P. Bertoli, et al., Automated composition of Web services by planning at the knowledge level, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2005, pp. 1252–1259.
[22] J. Hoffmann, P. Bertoli, M. Helmert, et al., Message-based Web service composition, integrity constraints, and planning under uncertainty: a new connection, J. Artif. Intell. Res. (JAIR) 35 (1) (2009) 49–117.
[23] J. Hoffmann, P. Bertoli, M. Pistore, Web service composition as planning, revisited: in between background theories and initial state uncertainty, in: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2007, pp. 1013–1018.
[24] J. Hoffmann, R. Brafman, Conformant planning via heuristic forward search: a new approach, Artif. Intell. (AIJ) 170 (6–7) (2006) 507–541.
[25] A. Tsalgatidou, T. Pilioura, An overview of standards and related technology in Web services, Distrib. Parall. Databases (DPD) 12 (2–3) (2002) 135–162.
[26] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufman Publishers, 2004.
[27] D. McDermott, M. Ghallab, A. Howe, et al., PDDL-the planning domain definition language, 1998.
[28] M. Klusch, A. Gerber, Fast composition planning of OWL-S services and application, in: Proceedings of European Conference on Web Services (ECOWS), 2006, pp. 181–190.
[29] J. Hoffmann, I. Weber, F. Kraft, Sap speaks PDDL: exploiting a software-engineering model for planning in business process management, J. Artif. Intell. Res. (JAIR) 44 (2012) 587–632.

[30] J. Hoffmann, I. Weber, F.M. Kraft, Planning@sap: an application in business process management, in: Proceedings of International Scheduling and Planning Applications Workshop (SPARK) at ICAPS, 2009.
[31] K. Currie, A. Tate, O-plan: the open planning architecture, Artif. Intell. (AIJ) 52 (1) (1991) 49–86.
[32] A. Tate, J. Dalton, O-plan: a common lisp planning web service, in: Proceedings of International Lisp Conference (ILC), 2003, pp. 12–25.
[33] S. Khan, K. Decker, W. Gillis, et al., A multi-agent system-driven ai planning approach to biological pathway discovery, in: Proceedings of International Conference on Automated Planning and Scheduling (ICAPS), 2003, pp. 246–255.
[34] S. Richter, M. Westphal, The lama planner: guiding cost-based anytime planning with landmarks, J. Artif. Intell. Res. (JAIR) 39 (1) (2010) 127–177.