Appl Intell (2014) 41:383–404
DOI 10.1007/s10489-014-0522-4

Towards automated choreography of Web services using planning in large scale service repositories

Guobing Zou · Yanglan Gan · Yixin Chen · Bofeng Zhang · Ruoyun Huang · You Xu · Yang Xiang

Published online: 15 March 2014
© Springer Science+Business Media New York 2014

Abstract Automated composition of Web services is becoming a prominent paradigm for implementing and delivering distributed applications. A composed service can be described either by orchestration or choreography. Service orchestration has a centralized controller which coordinates the services in a composite service. Differently, service choreography assumes that all of the participating services collaborate with each other to achieve a globally shared task. Choreography has received great attention and demonstrated a few key advantages over orchestration such as data efficiency, distributed control, and scalability. Although there is extensive research on the languages and protocols of choreography, automated design of choreography plans, especially distributed plans for multiple roles, is more complex and not studied before. In this paper, we propose a novel planning-based approach, including compilation of contingencies, stateful actions, dependency analysis and communication control, which can automatically convert a given composition task to a distributed choreography specification. The experimental results conducted on large scale service repositories show the effectiveness and efficiency of our approach for automated choreography of Web services.

Keywords Service choreography · Automated planning · Service composition · Service orchestration

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), and a National Science Foundation (IIS-0713109).

G. Zou · B. Zhang
School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China

G. Zou
e-mail: gbzou@shu.edu.cn

B. Zhang
e-mail: bfzhang@shu.edu.cn

Y. Gan (✉)
School of Computer Science and Technology, Donghua University, Shanghai 201620, China
e-mail: ylgan@dhu.edu.cn

Y. Chen · R. Huang · Y. Xu
Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA

Y. Chen
e-mail: ychen25@wustl.edu

R. Huang
e-mail: ruoyun.huang@wustl.edu

Y. Xu
e-mail: youxu@wustl.edu

Y. Xiang
Department of Computer Science and Technology, Tongji University, Shanghai 201804, China
e-mail: shxiangyang@tongji.edu.cn

1 Introduction

Web services are modular, self-describing and Web accessible distributed software components. They can be published on the Web, discovered by software agents and composed as new services with more complex functionalities. As the Service-Oriented Architecture (SOA) paradigm plays a key role in the development of enterprise application integration, Web services are becoming the most important fundamental building blocks for fast developing next generation applications. Despite the traditional Web service core standards

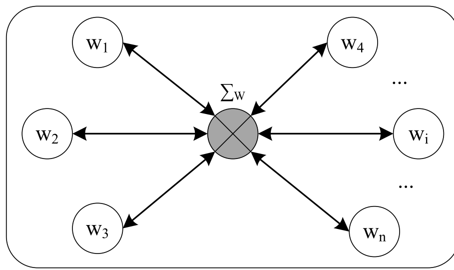


Fig. 1 The coordination model of service orchestration

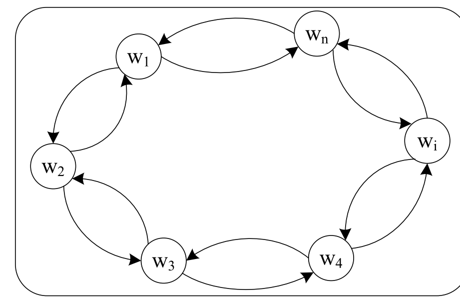


Fig. 2 The collaboration model of service choreography

(i.e., WSDL, SOAP, UDDI) support registry, discovery and consumption, in many cases there is no single Web service satisfying a given request.

The problem of combining a set of connected Web services together to create a more complex, value-added and cross-organizational business process is called *Web service composition* (WSC). It is designed and applied in the scenario, where no single service can be used to satisfy a service request. There are two ways for the description and execution of combining Web services, which are service *orchestration* and service *choreography*, respectively. That is, a WSC problem can be described from the view of a single participant by orchestration or a global perspective by choreography [2].

Service orchestration refers to an executable business process that has a central controller to coordinate all of the participating Web services [6, 25]. As shown in Fig. 1, the service orchestration model contains n participating services, each of which needs to communicate with the composite service by message exchanges, i.e., orchestrator Σ_W. In this way, it provides a mean to generate the internal executable behavioral business process of some specific service [5], which is responsible to coordinate the n services to complete a composition task. Business processes described by an orchestration language (e.g. BPEL4WS) can be executed on an orchestration engine, such as Active BPEL Open Engine [26]. Extensive researches based on service orchestration have been reported, such as continuous orchestration [3], dynamic service selection [16] and automated composition in asynchronous domains [4, 26, 27]. On the other hand, service choreography does not have an orchestrator, conversely all of the participating Web services collaborate with each other in order to achieve a shared goal. As illustrated in Fig. 2, the service choreography model tracks the message sequences among n independently autonomous services, rather than a specific business process that a single party executes [7, 22]. Typically, it concerns about distributed sequences and conditions. Thus, choreography is more collaborative and addresses the interactions that implement the collaboration among multiple services. Web Services Choreography Description Language (WS-CDL)[1] is an XML-based W3C candidate language for describing mutual collaboration of Web services.

Compared to service orchestration, a few advantages exist in Web service choreography from the perspective of large scale real-world applications: (1) data transfer efficiency. It requires less data transfer, compared to orchestration which requires large quantities of data exchanged between central controller and all participating services. (2) robustness. Choreography is more robust, since orchestration processes depend on a single central server whose failure may easily paralyze the whole system. (3) fairness. Choreography also provides a fair peer-to-peer model where each peer takes an equal role in collaboration, whereas in orchestration a central server has control over others. (4) designing multiple peers together in choreography also helps avoid deadlocks, which may occur when each peer executes its own orchestration plan.

Although service choreography is widely advocated, to the best of our knowledge, research efforts regarding automatic generation of service choreography specification have never been taken so far. WS-CDL describes choreography from a global view in a single master plan. We argue that choreography plans should be distributed. Although a global view (e.g., WS-CDL) is helpful, at the execution end, each role should have a "local plan", such as the Multiagent Protocols (MAP) [2] that specifies what it needs to do from its individual perspective. In this aspect, some recent research mainly focuses on new languages for distributed plans in choreography, but relies on users to manually generate the specifications. Examples include MAP [2] which can be used to describe a service choreography for multiple peers, and WS-CDL+ [17] which provides an extended execution engine for the enactment of the description of Web service choreography. They do not solve the problem of how to generate these choreography specifications, although there are works that verify the protocol using model checking [2, 30]. Due to the complexity of decentralized logics, manually developing such a specification for service choreography can

[1] http://www.w3.org/TR/ws-cdl-10/.

be a time-consuming, tedious and error-prone task, which is not appropriate for large-scale collaboration of Web services. Providing automated choreography of Web services is therefore essential to reduce the time to market of services, and ultimately to successfully enact the service-oriented approach.

Planning has been applied for automated service orchestration and ensures its correctness, but it is difficult and has not been applied for service choreography in several challenges. More specifically, while planning is suitable for constructing the composition plan from the view of a single party in orchestration, service choreography by definition needs distributed plans for multiple participants, making the problem more difficult for automated planning. Also, service choreography needs asynchronous communication support between peers and contingent plans that depend on the outcomes of services. Consequently, each service evolves independently with unpredictable speed, and collaborates with the other services only through asynchronous message exchanges. Therefore, how to automatically and efficiently build a service choreography has become a challenge and received considerable attention from both academia and industry.

In this paper, to address the above challenges, we propose a novel planning-based approach extended from our previous work [33] that automatically generates distributed choreography plans using automated planning. Our framework translates a set of available Web services, along with user-defined contingencies, into a planning domain in Planning Domain Definition Language (PDDL). This process compiles the contingency on action outcomes into a deterministic planning problem. Then, given a composition requirement task, a highly efficient automated planner is used to find a solution plan. Dependency analysis is performed on the solution plan to derive a dependency graph, which can be directly marked to provide a global view of the choreography master plan. We further propose a decentralization scheme which supports the synthesis of multiple local plans, one for each peer. Our decentralization scheme addresses three main features in choreography, including *choice*, *parallelism*, and *communication control*. As a result, our scheme ensures that all possible distributed execution sequences carried out by the participating peers are valid sequences under the centralized master plan, and collaborate together to solve a service choreography problem.

We implement a prototype system based on our scheme, and conduct extensive experiments on large scale Web service repositories containing 81,464 services collected from ICEBE05. The experimental results validate the feasibility of our work for service choreography. Comparison against other planning-based composition solvers shows that our approach also has superior efficiency in terms of solution plan generation.

The rest of this paper is organized as follows. In Sect. 2, we present a running example on a real world e-commerce application. Section 3 presents the problem formulation. In Sect. 4, we present our approach to automated service choreography using planning and dependency graph analysis. Section 5 gives the system architecture and implementation. Section 6 shows extensive experimental results on large scale Web service repositories. Section 7 reviews related work on Web service composition. Finally, Sect. 8 concludes the paper and discusses future work.

2 A running example

In this section, we initially give a motivating example in e-commerce application that will be used throughout the paper. Our running example includes three service roles for choreography: Customer, Supplier and Warehouse. Each service as a role provides a collection of functionalities by performing its operations. Specifically, the **Customer** inquires product information, sends a product request and makes a payment for product ordering. The **Supplier** focuses on receiving a purchase request, acquiring product availability and confirming an order status. The **Warehouse** checks the availability status of a product as requested and provides the shipping of a product order.

The goal is to implement a composed service for product purchasing and delivering by composing independent existing service roles. These service roles collaborate with each other to achieve a situation where either the **Customer** service role is successfully provided with a given product from the **Supplier** service role, or the product ordering fails due to product unavailability from the **Warehouse** service role. Figure 3 illustrates detailed process of product ordering among three choreography roles.

The **Customer** sends a quote request to the **Supplier** on a given product, and then the **Supplier** receives the request (ReceiveRFQ) and replies. After receiving it (ReceiveQuote), the **Customer** sends a product order request to the **Supplier**. Once receiving an order (ReceivePO), the **Supplier** sends order information to the **Warehouse**, who receives it (ReceiveOI), checks its availability (CheckAvail), and replies to the **Supplier**. When **Supplier** receives it (ReceiveAvail) from the **Warehouse**, it makes a decision for the request.

There are two possibilities. If a product order is unavailable, the **Supplier** cancels it (CancelPO) and notifies the **Customer** with an order rejection confirmation. After the **Customer** receives it (ReceivePOReject), the order process terminates. Otherwise, the **Supplier** accepts it (ConfirmPO). In such a case, the **Supplier** replies an order acceptance to the **Customer** and sends a shipping order

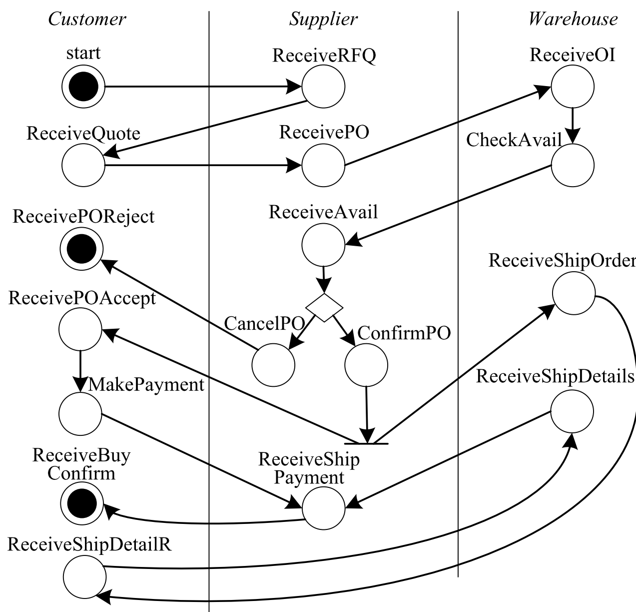


Fig. 3 The choreography interaction flow of service roles

request to the **Warehouse**. After receiving acceptance (ReceivePOAccept), the **Customer** makes a payment (MakePayment) and replies a payment confirmation to the **Supplier**. Meanwhile, after receiving the shipping order request (ReceiveShipOrder), the **Warehouse** sends a request to the **Customer** for shipping details. Once the **Customer** receives it (ReceiveShipDetailR), it replies details to the **Warehouse**. After receiving it (ReceiveShipDetails), the **Warehouse** replies a shipping confirmation to the **Supplier**. Finally, when a payment confirmation from the **Customer** and a shipping confirmation from the **Warehouse** are both received (ReceiveShipPayment), the **Supplier** replies a purchase success confirmation to the **Customer**, who receives it (ReceiveBuyConfirm) and the product order process terminates.

Notice that here we assume multiple participating services exist in a service composition problem, and we mainly demonstrate the mutual interactions among different service roles involved in service choreography. However, we will elaborate how to automatically compose these service roles and then generate distributed choreography specification using planning and dependency graph analysis in the subsequent sections.

3 Problem formulation

We first formulate our problem using a simplified model. In this paper, we focus on understanding the fundamental principles of automated choreography of Web services by planning, and simplify or omit certain issues in Web service composition, such as ontology [1] and background theory [13]. It is our future work to integrate the theory and algorithms with other techniques into more advanced systems.

Definition 1 (Web Service) Web service w consists of a set of operations, denoted as $w = \{op_1, op_2, \ldots\}$, where $\forall op \in w$ is a 2-tuple $\langle I, O\rangle$, $I = \{I^1, I^2, \ldots\}$ is a set of input interface parameters. Similarly, $O = \{O^1, O^2, \ldots\}$ is a set of output interface parameters. We use $op.I$ and $op.O$ to denote I and O in op, respectively. For each interface parameter x, we use $Dom(x)$ to denote its possible values and $x.value \in Dom(x)$ to denote the value of x.

Note that each Web service plays a role that can perform a set of operations. A service repository is a set of services.

Definition 2 (Contingency) Given an operation op with its input interface parameters $op.I = \{I^1, I^2, \ldots\}$, a contingency is a tuple $c = (op, I^i, pre^i)$ where $pre^i \subset Dom(I^i)$.

A contingency $c = (op, I^i, pre^i)$ means that to invoke op, we need its input parameter I^i to take the values in pre^i, instead of any value in $Dom(I^i)$.

We define a **service state** as a set of interface parameters $Q = \{x^1, x^2, \ldots\}$. We assume that parameters not in Q are unavailable at the state.

Definition 3 (Applicability) Without contingency, an operation op is applicable at a service state Q if $op.I \subseteq Q$. We denote this as $Q \succ op$. An operation op under contingency $c = (op, I^i, pre^i)$ (denoted as $c \rhd op$) is applicable at Q if $op.I \subseteq Q$ and $op.I^i.value \in pre^i$. We denote this as $Q \succ c \rhd op$.

When an applicable operation op or $c \rhd op$ is applied to Q, the resulting state $Q' = Q \oplus op$ (or $Q' = Q \oplus c \rhd op$) is $Q' = Q \cup op.O$. An **execution sequence** is an ordered list $L = (o_1, \ldots, o_m)$, where each element is either an operation op or an operation with contingency $c \rhd op$. Applying a sequence L to a service state Q results in $Q' = Q \oplus L = (\cdots((Q \oplus o_1) \oplus o_2) \cdots \oplus o_m)$ if every step is applicable (otherwise $Q \oplus L$ is undefined).

Definition 4 (Choreography Request) A choreography request, r, is a 2-tuple $\langle r_{in}, r_{out}\rangle$, where $r_{in} = \{r^1_{in}, r^2_{in}, \ldots\}$ is an initial interface parameter set provided as request inputs, and $r_{out} = \{r^1_{out}, r^2_{out}, \ldots\}$ is a goal interface parameter set desired to be returned to the users.

Given a set of Web services and a service choreography request, we define a service choreography problem as below.

Definition 5 (Service Choreography Problem) A service choreograph problem (SCP) is defined by a 4-tuple (W, C, r_{in}, r_{out}). Where,

(1) $W = \{w_1, \ldots, w_N\}$ is a service repository;
(2) $C = \{c_1, \ldots, c_{N_c}\}$ is a set of contingencies;
(3) $r_{in} = \{r^1, r^2, \ldots\}$ is an input parameter set;
(4) $r_{out} = \{q^1, q^2, \ldots\}$ is an output parameter set.

Example 1 Following our running example in Sect. 2, its SCP is represented as below. $W = \{Customer, Supplier, Warehouse\}$, where $Supplier = \{ReceiveRFQ, ReceivePO, ConfirmPO, \ldots\}$. An operation in *Supplier* is $ReceiveRFQ = \langle I, O\rangle$, where $I = \{pid, pid_name\}$, and $O = \{pid_price\}$. Another operation is $ConfirmPO = \langle I, O\rangle$, where $I = \{po_avail\}$, $O = \{po_accept, shiporderR\}$. $Dom(po_avail) = \{avail, not_avail\}$, while *po_accept* and *shiporderR* represent an order acceptance and shipping order request, respectively.

The contingency set consists of $C = \{c_1, c_2\}$, where $c_1 = (ConfirmPO, po_avail, \{avail\})$ and $c_2 = (CancelPO, po_avail, \{not_avail\})$.

The request inputs include three interface parameters, $r_{in} = \{pid, pid_name, pid_quantity\}$. Goal specification is $r_{out} = \{purchase_confirm, po_order_reject\}$, which has *purchase_confirm* for success purchase confirmation and an order rejection *po_order_reject*.

A user can specify multiple possible goal states desired by the business process of composition request. Typically, the users are knowledgeable of these multiple choreography goals, because they are advanced model developers of Web service composition. A SCP should consider all of the contingencies and give service choreography plans that can handle the various goals. In our example, the user specifies both $purchase_confirm$ and po_order_reject as goals, so that the service choreographer can find plans contingent on the availability of the product.

Given a SCP $= (W, C, r_{in}, r_{out})$, **choreography master plan** is any expression P defined by the language.

$$
\begin{array}{lll}
P ::= op & & (op \in w \in W) \\
\quad \mid & talk(i, j) & (role\ i\ talks\ to\ j) \\
\quad \mid & P; P & (sequential) \\
\quad \mid & P \parallel P & (parallel) \\
\quad \mid & c \rhd P \text{ or } P & (c \in C, choice)
\end{array}
$$

where *op* is an operation in a Web service and *c* is a contingency. We allow only one contingency for each or choice to simplify the presentation, although it is easy to extend to multiple contingencies. Our language is similar to other choreography description models [2, 28], except that we explicitly introduce contingency in our definition. The *talk* action is applicable to any service state and brings no change to the state.

Given a service state Q, let $t(P)$ denote all of the possible execution sequences from Q, we can define:

$$
\begin{aligned}
&t(op) = \{(op)\} \\
&t(talk(i, j)) = \{(talk(i, j))\} \\
&t(P_1; P_2) = t(P_1) \circ t(P_2) \\
&t(P_1 \parallel P_2) = t(P_1) \bowtie t(P_2) \\
&t(c \rhd P_1 \text{ or } P_2) = \begin{cases} t(P_1), & \text{if } Q \succ c \rhd P_1; \\ t(P_2), & \text{otherwise.} \end{cases}
\end{aligned}
$$

where $\circ$ denotes the concatenation of two sequence sets (i.e. $A \circ B = \{(a, b) | a \in A, b \in B\}$), and $\bowtie$ is the interleaving of two sequence sets; $Q \succ c \rhd P_1$ should be understood as $Q \succ c \rhd op$ for any *op* that can be the first operation in a sequence in $t(P_1)$.

Given any expression P, the function $t(P)$ denotes the combination of the sequences that can be drawn from the P. As a result, starting from a service state we apply each sequence in $t(P)$ and merge their execution states as one, which can reach a desired goal state. It is defined as below.

Definition 6 (Centralized Solution) A centralized solution to a SCP $= (W, C, r_{in}, r_{out})$ is a choreography master plan P, such that for every sequence $L_k \in t(P)$, $r_{in} \oplus L_k$ is defined and $\bigcup_{L_k \in t(P)} (r_{in} \oplus L_k) \supseteq r_{out}$.

In SCP, there are multiple roles and each corresponds to a $w \in W$. The philosophy of service choreography is to let each role execute a local plan so that the multiple roles collaborate and finish a global task. A **local plan** is any expression R defined by the language.

$$
\begin{array}{lll}
R ::= op & & (op \in w_i) \\
\quad \mid & send(ch, i, j) & (send\ to\ role\ j) \\
\quad \mid & recv(ch, j, i) & (receive\ from\ role\ j) \\
\quad \mid & R; R & (sequential) \\
\quad \mid & R \parallel R & (parallel) \\
\quad \mid & c \rhd R \text{ or } R & (c \in C^i, choice)
\end{array}
$$

where $C^i \subseteq C$ is the set of contingencies related to w_i (i.e. C^i includes those c whose operation *op* is in w_i); and *ch* is a unique communication channel ID for each $send/recv$ pair. Like *talk*, *send* and *recv* are applicable to any service state.

Given a service state Q, for role w_i with a local plan R_i, we define $t(R_i)$, the set of possible execution sequences of w_i from Q. A **distributed choreography plan** R is a set of local plans R_i, one for each role w_i. Then, the set of **combination sequences** is

$$
\mathbf{C}(R) = \{\bowtie^* (L_1, \ldots, L_N) | L_i \in t(R_i), i = 1..N\}
$$

where $\bowtie^*$ denotes any interleaving of N sequences subject to one constraint: $send(ch, i, j)$ is always sent before $recv(ch, j, i)$ for any ch, i and j.

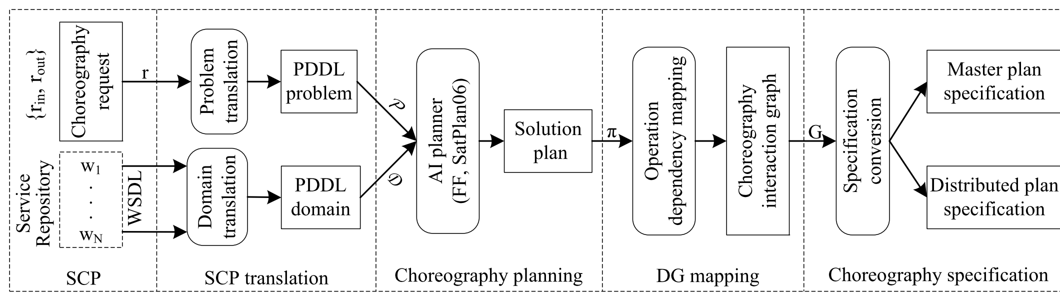


Fig. 4 The approach of automated choreography of Web services using planning

Definition 7 (Distributed Solution) A distributed solution to a SCP $= (W, C, r_{in}, r_{out})$ is a distributed choreography plan R, such that for every sequence $L_k \in \mathbf{C}(R)$, $r_{in} \oplus L_k$ is defined and $\bigcup_{L_k \in \mathbf{C}(R)} (r_{in} \oplus L_k) \supseteq r_{out}$.

Definition 8 (Equivalence) A centralized solution shows equivalence to a distributed one when their sequence sets contain identical sequences, ignoring *talk*, *send*, and *recv*.

A closely related equivalence has been studied in [28]. We comment that WS-CDL can be viewed as an extended language for choreography master plan, while MAP [2] and the Role Language in [28] are examples of languages for distributed plans.

4 Automated choreography by planning

We develop an approach that can correctly generate a distributed solution for a SCP. Figure 4 illustrates an overview of how we solve a SCP. It has a few major steps. (1) Translate a SCP into a PDDL planning problem, which complies action contingencies. (2) Solve the planning problem using an automated planner to obtain a solution plan. (3) Perform a dependency analysis on the solution plan to build a choreography dependency graph (DG). (4) Mark the DG using node out degree to generate a master plan P. (5) Project P to a distributed plan R based on the DG.

4.1 SCP translation

In this section, we translate a SCP into a classical planning problem. As a classical planning task is defined by a planning domain and a planning problem, the task of translating a SCP corresponds to domain translation and problem translation. We use the Planning Domain Definition Language (PDDL)[2] to describe a choreography planning problem.

[2]PDDL is an action-centered description language that is inspired by STRIPS formulations of AI planning problems and widely used for describing classical planning tasks.

Definition 9 (Choreography Planning Problem) In a SCP $= (W, C, r_{in}, r_{out})$, SCP translation transforms it into a choreography planning problem, denoted as $\langle \mathcal{D}, \mathcal{P} \rangle$, which has a choreography domain $\mathcal{D}$ and a choreography problem $\mathcal{P}$, where, $\mathcal{D}$ and $\mathcal{P}$ are as follows.

1. $\mathcal{D} = (\mathcal{T}, \mathcal{S}, \mathcal{A})$, where $\mathcal{T}, \mathcal{S}, \mathcal{A}$ are types, predicates and actions in a PDDL choreography domain $\mathcal{D}$.
2. $\mathcal{P} = (\mathcal{O}, s_o, g)$, where $\mathcal{O}, s_o, g$ are objects, initial state and goal state in a PDDL choreography problem $\mathcal{P}$.

As described above, given a SCP $= (W, C, r_{in}, r_{out})$, domain translation converts the service repository W and contingencies C into a choreography domain $\mathcal{D}$ in PDDL. Problem translation is responsible to translate a choreography request $\langle r_{in}, r_{out} \rangle$ into a choreography problem $\mathcal{P}$ in PDDL.

4.1.1 *Choreography domain translation*

The core process for domain translation, listed in Algorithm 1, models an operation $op \in w$ (or a contingency $c \in C$) as an **operation action** a (or a **contingency action** b_c). Each action has a set of preconditions $pre(a)$ and effects $eff(a)$. For each action $a \in \mathcal{A}$, we also define two properties: $key(a)$ denotes the operation or contingency that a is representing; and $host(a)$ denotes the Web service w that $key(a)$ belongs to.

In SCP translation, to simplify problem expression, we assume that all of the input and output parameters in an operation have a uniform type $\mathtt{string}$, and all of the precondition and effect propositions in an action are used to express availability of a parameter by a general predicate $(yes\ ?p)$. Although we only allow a simple data type and a predefined predicate for the presentation of service interface parameters, it is easy to extend our approach to deal with situations where there are complex data types involved and multiple predicates. The reason is that to find a choreography solution plan for a deterministic planning problem, as described in Definition 9, most of existing off-the-shelf AI planners (e.g., FF [15]) apply heuristic search algorithm for the matching of effect and precondition propositions between two actions with logic reasoning techniques. They can accept multiple

Algorithm 1: choreography_domain_translation

Input: service repository W; a set of contingencies C;
Output: a choreography domain $\mathcal{D}$; facts $\mathcal{F}$ to be used as objects in $\mathcal{P}$;

```
1  let D(T, S, A), F be empty sets;
2  T ← {string}; S ← {(yes ?p)};
3  assign actions a ← ⟨{}, {}⟩; b_c ← ⟨{}, {}⟩;
4  foreach w ∈ W do
5      let OP be the set of operations in w;
6      foreach op ∈ OP do
7          key(a) ← op; host(a) ← w;
8          if ∃c ∈ C, c = (op, I^i, pre^i) then
9              key(b_c) ← c; host(b_c) ← w;
10             pre(b_c) ← {(yes I^i)};
11             eff(b_c) ← {(yes cont_c)};
12             foreach I^j ∈ op.I \ {I^i} do
13                 pre(a) ← pre(a) ∪ {(yes I^j)};
14             pre(a) ← pre(a) ∪ {(yes cont_c)};
15             foreach O^j ∈ op.O do
16                 eff(a) ← eff(a) ∪ {(yes O^j)};
17             A ← A ∪ {a} ∪ {b_c};
18             F ← F ∪ op.I ∪ op.O ∪ {cont_c};
19         else
20             foreach I^j ∈ op.I do
21                 pre(a) ← pre(a) ∪ {(yes I^j)};
22             foreach O^j ∈ op.O do
23                 eff(a) ← eff(a) ∪ {(yes O^j)};
24             A ← A ∪ {a};
25             F ← F ∪ op.I ∪ op.O;
26         a ← ⟨{}, {}⟩; b_c ← ⟨{}, {}⟩;
27 return D, F;
```

data types and first-order predicates. However, when service providers publish services in a repository, they need to specify input and output parameters with our provided data types and multiple predicates. By doing so, our approach can automatically translate Web services into a choreography domain without any manual deployment.

Based on above assumption, the domain translation procedure in Algorithm 1 works as follows. First, we assign the data type set $\mathcal{T}$ as $\{string\}$ for the input and output parameters in an operation or a contingency (Line 2), which is required to represent a choreography domain in PDDL fed into an automated planner to find a solution plan. Then, for each service $w \in W$, we translate each of its operations $op \in w$ (or a contingency c) into an operation action $a \in \mathcal{A}$ (or a contingency action b_c) (Lines 4–26). For each operation $op \in w$, we first set its operation action a with $key(a) = op$ and $host(a) = w$ (Line 7). Then, there are two possibilities to model an action for the operation op.

(1) If op has a contingency $c \in C$, $c = (op, I^i, pre^i)$ (Lines 8–18), we first introduce a **contingency action** b_c and define $key(b_c) = c$, $host(b_c) = w$, $pre(b_c) = \{(yes\ I^i)\}$, and $eff(b_c) = \{(yes\ cont_c)\}$, where $op \in w$ (Lines 9–11). Here, $cont_c$ is a special fact introduced for each contingency c. We also define an **operation action** a for op by $pre(a) = \{(yes\ I^j) \mid I^j \in op.I, I^j \neq I^i\} \cup \{(yes\ cont_c)\}$ and $eff(a) = \{(yes\ O^j) \mid O^j \in op.O\}$ (Lines 12–16). After a and b_c are modeled, we put them into the action set $\mathcal{A}$, and aggregate all of the input and output parameters ($op.I$ and $op.O$) and the special contingency fact $cont_c$ together into $\mathcal{F}$ (Lines 17–18), which is used as objects in a choreography problem $\mathcal{P}$.

(2) If op does not have a contingency $c \in C$ (Lines 19–25), we only define an **operation action** a for op by $pre(a) = \{(yes\ I^j) \mid I^j \in op.I\}$ and $eff(a) = \{(yes\ O^j) \mid O^j \in op.O\}$ (Lines 20–23). Then, we put a into $\mathcal{A}$ and its parameters into $\mathcal{F}$ (Lines 24–25).

Example 2 By the Algorithm 1, the SCP shown in Example 1 is translated into a choreography domain $\mathcal{D}$ with $\mathcal{A} = \{ReceiveRFQ, ConfirmPO, \ldots, b_{c1}, b_{c2}\}$, which includes 18 actions (16 operation actions and 2 contingency actions). For action a from $ReceiveRFQ$, $pre(a) = \{(yes\ pid), (yes\ pid_name)\}$, $eff(a) = \{(yes\ pid_price)\}$. For contingency $c_1 = (ConfirmPO, po_avail, \{avail\})$, its translated action b_{c1} has $pre(b_{c1}) = \{(yes\ po_avail)\}$ and $eff(b_{c1}) = \{(yes\ cont_{c1})\}$. Moreover, the action a' for $ConfirmPO$ has $pre(a') = \{(yes\ cont_{c1})\}$ and $eff(a') = \{(yes\ po_accept), (yes\ shiporderR)\}$. Finally, we get facts $\mathcal{F} = \{pid, pid_name, \ldots, cont_{c1}, cont_{c2}\}$.

4.1.2 Choreography problem translation

Based on a set of facts $\mathcal{F}$ extracted from the input and output parameters of operations, including all of the special contingency facts, Algorithm 2 translates a choreography request into a PDDL problem $\mathcal{P}$.

The procedure of Algorithm 2 works as follows. We first take each fact $f \in \mathcal{F}$ as a problem object in $\mathcal{O}$ (Line 2). Then, for each request input $r^i_{in} \in r_{in}$, we apply predicate $(yes\ ?p)$ to generate an initial state proposition $(yes\ r^i_{in})$. The conjunction of all the propositions produced by r_{in} makes an initial state $s_0 = \{(yes\ r^i_{in}) \mid r^i_{in} \in r_{in}\}$ (Lines 3–5). Finally, we apply $(yes\ ?p)$ to each goal parameter $r^i_{out} \in r_{out}$ for generating a proposition set $\{(yes\ r^i_{out})\}$ as goal specification. As a result, $g = \{(yes\ r^i_{out}) \mid r^i_{out} \in r_{out}\}$ (Lines 6–8).

Example 3 By the problem translation of Algorithm 2, the SCP shown in Example 1 is translated into a choreography problem $\mathcal{P}$ with a set of objects $\mathcal{O} = \{pid, pid_name,$

Algorithm 2: choreography_problem_translation

Input: a choreography request: r_{in} and r_{out}; a set of facts $\mathcal{F}$ as objects;

Output: a PDDL choreography problem $\mathcal{P}$;

```
1 let P(O, s_o, g) be empty sets;
2 O ← O ∪ F;
3 foreach r^i_in ∈ r_in do
4     apply (yes ?p) to r^i_in, generate (yes r^i_in);
5     s_o ← s_o ∪ {(yes r^i_in)};
6 foreach r^i_out ∈ r_out do
7     apply (yes ?p) to r^i_out, generate (yes r^i_out);
8     g ← g ∪ {(yes r^i_out)};
9 return P;
```

$\ldots, cont_{c1}, cont_{c2}\}$, an initial state $s_0 = \{(yes\ pid), (yes\ pid_name), (yes\ pid_quantity)\}$, and $g = \{(yes\ purchase_confirm), (yes\ po_order_reject)\}$.

4.2 Time complexity analysis of SCP translation

Let SCP $= (W, C, r_{in}, r_{out})$ be a Web service choreography problem, where $W = \{w_1, \ldots, w_N\}$ is a repository with N number of Web services, $C = \{c_1, \ldots, c_{N_c}\}$ is a set of contingencies, $r_{in} = \{r^1, r^2, \ldots\}$ is a set of input parameters that are provided as initial conditions, and $r_{out} = \{q^1, q^2, \ldots\}$ is a set of output parameters as desired goal specification.

The choreography planning formulation of SCP is composed of a domain translation and a problem translation. The former part models each operation $op \in w$ or contingency $c \in C$ as an operation action a or a contingency action b_c that leads to a choreography domain $\mathcal{D}$. The latter translates $\langle r_{in}, r_{out}\rangle$ together with facts $\mathcal{F}$ to a choreography problem $\mathcal{P}$.

The computational complexity of generating a choreography domain is determined by mapping operations or contingencies into actions as well as problem objects. Its time complexity is calculated by $T_{domain} = O(\sum_{w\in W}\sum_{op\in w}(2 + |C| + 4 + |op.I| + 1 + |op.O| + 1 + |op.I| + |op.O| + 1))$, where $|C|$ is the number of contingencies in W. For each $op \in w \in W$, we denote the number of input and output parameters as $K^I_{op} = |op.I|$ and $K^O_{op} = |op.O|$, respectively. Suppose that $K = \max_{op\in w}\{|op.I| + |op.O|\}$ is an upper bound on the number of input and output parameters among all the operations in W. We use M to denote the maximum number of operations involved in each service within a repository. By the replacement with above parameters, the time complexity of choreography domain translation can be recalculated by $T_{domain} = O(\sum_{w\in W}\sum_{op\in w}(N_c + 2*(K^I_{op} + K^O_{op}) + 9)) = O(N*M*(2K + N_c + 9)) = O(N*M*(K + N_c))$.

The time cost of choreography problem translation is dominated by three parts: the number of objects in facts $\mathcal{F}$, the size of initial and goal parameters in a request $\langle r_{in}, r_{out}\rangle$. Considering the worst case, no repeated input and output interface parameters exist among Web services. Thus, the time complexity of problem translation is $T_{problem} = O(\sum_{w\in W}\sum_{op\in w}(|op.I| + |op.O|) + |C| + |r_{in}| + |r_{out}|) = O(\sum_{w\in W}\sum_{op\in w}(|K^I_{op}| + |K^O_{op}|) + N_c + |r_{in}| + |r_{out}|) = O(N*M*K + N_c + |r_{in}| + |r_{out}|)$. In terms of a large scale Web service repository, since we have $N_c \ll N$, $|r_{in}| \ll N$, and $|r_{out}| \ll N$, the computational complexity of choreography problem translation is $T_{problem} = O(N*M*K)$.

From the above computational analysis, we can see that for a large scale service repository where we have $N \gg M$, $N \gg K$, and $N \gg N_c$, our approach of SCP translation is almost a linear time algorithm with respect to the number of services in W. Thus, a SCP can be efficiently performed and translated into a choreography planning problem in polynomial time.

4.3 Finding a solution plan

Given a SCP $= (W, C, r_{in}, r_{out})$, we translate it into a $\langle\mathcal{D}, \mathcal{P}\rangle$. Then, we use an AI planner to automatically find a solution plan that can transform the initial state s_0 to an end state S such that $g \subseteq S$.

Given a planning state $X = \{(yes\ x_1), (yes\ x_2), \ldots\}$, an operation action a (or a contingency action b_c) can be applicable to X, if $pre(a) \subseteq X$. Again, we denote it as $X \oplus a = X \cup eff(a)$.

Definition 10 (Choreography Planning Satisfiability) Given two planning states $X = \{(yes\ x^1), (yes\ x^2), \ldots\}$, $Y = \{(yes\ y^1), (yes\ y^2), \ldots\}$, and a set of actions $\mathcal{A}$, if $X \oplus a_i \oplus \cdots \oplus a_j \supseteq Y$, $1 \le i, j \le |\mathcal{A}|$, we say $(a_i, \ldots, a_j)$ is a solution plan sequence and denote this relationship as $(a_i \otimes \cdots \otimes a_j) \propto (X \to Y)$.

Definition 11 (Solution Plan) Let $\langle\mathcal{D}, \mathcal{P}\rangle = (\mathcal{D}, s_0, g)$ be a choreography planning problem, a solution plan is a sequence of actions, $\pi = (a_1, \ldots, a_m)$, such that $(a_1 \otimes \cdots \otimes a_m) \propto (s_0 \to g)$ is satisfiable.

Any automated planner that supports PDDL 1.0 and up can be applied to solve the choreography planning problem. Currently, we adopt two planners: FF [15] and SatPlan06 [19].

We analyze the time complexity of generating a solution plan using FF, which is one of the most successful automatic planners. It utilizes heuristic search strategy called relaxed GraphPlan to find a non-optimal plan for a given deterministic planning problem. In almost all of the existing benchmark domains, it can be proven to solve relaxed tasks in

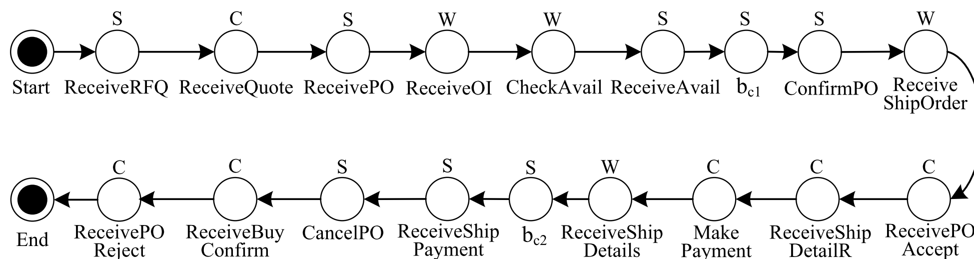


Fig. 5 The solution plan for the running example. The host of each action is also added on top of the action. (C: Customer; S: Supplier; W: Warehouse)

polynomial time and work well empirically on a large class of planning tasks.

Example 4 Reconsider our SCP in the running example. After SCP translation, we take $\langle \mathcal{D}, \mathcal{P} \rangle$ shown in Examples 2 and 3 as input, and then use FF [15] to find a solution plan. Figure 5 illustrates the solution plan. As described in Definition 10, it fulfills choreography planning satisfiability: $(ReceiveRFQ \otimes ReceiveQuote \otimes \cdots \otimes b_{c1} \otimes \cdots \otimes ReceivePOReject) \propto (s_0 \rightarrow g)$, where $s_0 = \{(yes\ pid), (yes\ \ pid_name), (yes\ \ pid_quantity)\}$, and $g = \{(yes\ purchase_confirm), (yes\ po_order_reject)\}$.

4.4 Constructing choreography dependency graph

From a solution plan $\pi = (a_1, \ldots, a_m)$, we can extract a choreography dependency graph based on dependency analysis between actions.

Definition 12 (Dependency) Given a solution plan $\pi = (a_1, \ldots, a_m)$, an action a_j depends on a_i (denoted as $a_i \vdash a_j$) if and only if $i < j$ and there exists at least a fact $f \in pre(a_j)$, such that $f \notin s_0$ and a_i is the last action in $a_1, \ldots, a_{j-1}$ and $f \in eff(a_i)$.

Dependency is general for both operation actions and contingency actions, and it can exist between two actions from the same or different service roles. By using dependency relationship among actions, we draw a dependency graph as below.

Definition 13 (Dependency Graph (DG)) Given a solution plan $\pi = (a_1, \ldots, a_m)$, a dependency graph is a directed graph $G = (V, E)$ such that $V = \pi$ and there is an edge $(a_i, a_j) \in E$ if and only if $a_i \vdash a_j$.

Intuitively, a dependency graph describes a choreography model encompassing interactions among the actions of multiple collaborative roles. Each vertex in G is an operation action a or a contingency action b_c. Each edge (a_i, a_j) in G represents an interaction activity, in which the role of action a_i sends messages to the role of action a_j, so that the action a_j can be performed after receiving the messages from a_i.

Algorithm 3: choreography_dependency_graph

Input: a solution plan π; an initial state s_0; a goal specification g;
Output: a choreography dependency graph G;

```
1 G(V) ← π;
2 G(E) ← {};
3 foreach a ∈ G(V) do
4 |   visited[a] ← false;
5 choreography_edges(s0, Start, {}, {});
6 return G;
```

Given a solution plan π, an initial state s_0 and a goal state g, the choreography_dependency_graph() in Algorithm 3 generates a dependency graph G.

In Algorithm 3, all actions are directly collected from π as the vertex set $G(V)$. Then, for each vertex a in $G(V)$, we mark it unvisited. Starting from the initial state s_0, we call a recursive procedure choreography_edges() to generate a choreography graph edges set $G(E)$. An artificial action *Start* to serve as the beginning action is created in the dependency graph.

Each invocation to procedure choreography_edges() (as shown in Algorithm 4) works as follows. We first choose a set of actions applicable to the current planning state S and put them into F (Line 2). Specifically, for each action a that is involved in $G(V)$, if it is not visited and its preconditions can be satisfied by S (i.e., $pre(a) \subseteq S$), we put it into F as an applicable action (see Algorithm 5). After choosing a set of applicable actions in F, there are two possibilities.

(1) If F has at least one applicable action, for each action $a_r \in F$, we first update the current planning state S as S_u by adding the effects of a_r, mark a_r as a visited action, create a new edge into $G(E)$ from a_l to a_r (Lines 3–7), and recursively call choreography_edges() using updated state S_u and action a_r (Line 8).
(2) The second possibility is that there is no action applicable to the current planning state S. In such case, we first update combination planning state S' using the current state S and append a_l to combination actions set A' (Line 10). Then, we search for an applicable action using S' (Line 11). If no such action can be found

Algorithm 4: choreography_edges

```
Input: a planning state S; a left vertex a_l used to create
       a new edge; a combination planning state S'; a
       set of combination actions A';
Output: a graph edges set G(E);
1  if S ∩ g ≠ {} then return;
2  F ← choose_actions(S);
3  if F has applicable actions then
4      foreach a_r ∈ F do
5          S_u ← S ∪ eff(a_r);
6          visited[a_r] ← true;
7          G(E) ← G(E) ∪ {(a_l, a_r)};
8          choreography_edges(S_u, a_r, S', A');
9  else
10     S' ← S' ∪ S; A' ← A' ∪ {a_l};
11     F' ← choose_actions(S');
12     if F' = {} then return;
13     else if F' has an applicable action a_c then
14         S_u ← S' ∪ eff(a_c);
15         visited[a_c] ← true;
16         foreach a' ∈ A' do
17             G(E) ← G(E) ∪ {(a', a_c)};
18         S' ← {}; A' ← {};
19         choreography_edges(S_u, a_c, S', A');
```

Algorithm 5: choose_actions

```
Input: a planning state S;
Output: a set of actions F applicable to S;
1  F ← {};
2  foreach a ∈ G(V) do
3      if visited[a] = false and pre(a) ⊆ S then
4          F ← F ∪ {a};
5  return F;
```

(Line 12), we return to the previous planning state S and choose another applicable action to recursively find graph edges (Lines 4–8). On the contrary, if we find an action a_c applicable to S', we first update S_u using S' and the effects of a_c, and mark a_c as a visited action (Lines 14–15). Then, we create a set of new edges from each combination action a' to a_c (Lines 16–17). Finally, we clear S' and A' and recursively call choreography_edges() with updated state S_u and left vertex a_c (Lines 18–19).

Given a solution plan, $\pi = (a_1, \ldots, a_m)$, it contains m operation or contingency actions. The time complexity of mapping from π to a dependency graph G is dominated by the generation of vertices and edges. Thus, we have $T_{DG} = O(m + T_{edges})$, where $O(m)$ is the cost of the construction of vertices with unvisited marking, and T_{edges} denotes the time complexity of creating graph edges. Considering the special case, where l actions are found for edge expansion in each recursive process, so we have $T_{edges}(m) = l * T_{edges}(m/l) + O(m * K)$, where K is the maximum number of precondition and effect propositions in an action. Suppose that $m = l^k$ and $T_{edges}(m) = T_{edges}(l^k) = h(k)$, then we transfer the complexity to $h(k) = l * h(k-1) + K * l^k$. After the recursive computation, we have $h(k) = K * l^k + K * l^k * k$. That is, $T_{edges} = O(K * m + K * m * \log_l^m)$ by replacing l^k and k with m and $\log_l^m$, respectively. As a result, the time complexity of DG generation is $T_{DG} = O(m + K * m + K * m * \log_l^m) = O(K * m * \log_l^m)$. Moreover, the expansion factor l and the number of actions m in π are considerably small compared to a large scale service repository, i.e., $l \ll N$ and $m \ll N$. Therefore, it is extremely efficient in dependency graph generation.

Example 5 Taking the solution plan π (shown in Fig. 5), the initial state s_0 and goal specification g (specified in Example 3) as inputs, we apply above three algorithms to generate a dependency graph G. Figure 6 shows the dependency graph, in which three roles collaborate with each other to complete a product order process. The dependency graph discovers multiple control flow structures, including sequential, parallel and conditional. We observe that there is a choice based on contingency actions (b_{c1} and b_{c2}) after the Supplier receives the availability information of an order. Moreover, once an order is accepted (*ConfirmPO*), the three service roles work in parallel to make payment and arrange shipping.

4.5 Generating the master choreography plan

A choreography master plan can be derived from a dependency graph (DG). Here, the following are the main rules.

1. For every operation action a whose out degree is 1, we mark a by the sequential sign ";".
2. For every contingency action b_c in the DG, we mark it by the contingency sign "$\triangleright$".
3. For every action a whose out degree is more than 1, we mark a by the parallel sign "$\|$" if its successors do not include contingency actions and mark a by the choice sign "or" otherwise.
4. For every edge (a_i, a_j), if $host(a_i) \neq host(a_j)$, we mark the edge by $talk(i, j)$.

After marking a DG according to above rules, we can write out the choreography master plan by viewing the marked DG as the parsing graph for the master plan language.

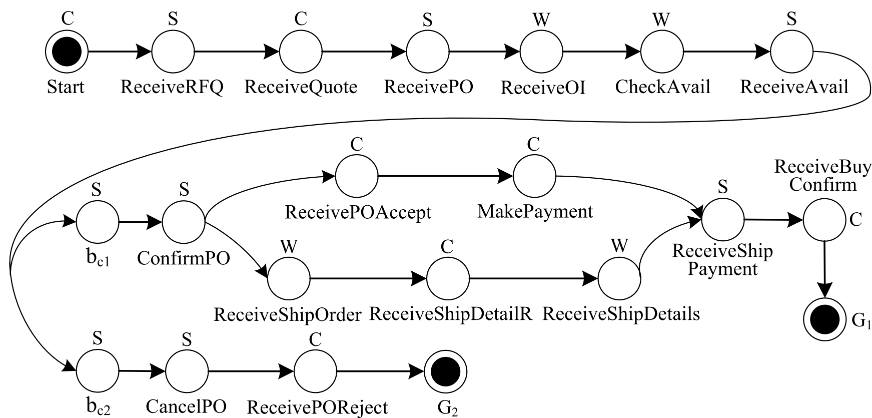


Fig. 6 The choreography dependency graph for the solution plan in Fig. 5. The goal states (G_1 and G_2) are also added to the graph for better illustration

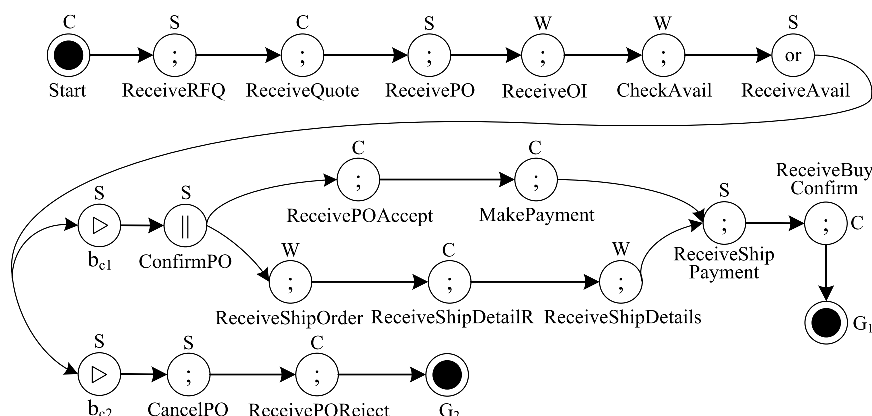


Fig. 7 The choreography master plan derived from the DG for Example 5

For the generation of choreography master plan, the cost is dominated by the traversal of a dependency graph DG. Considering the linked table as the representation of DG, the time complexity is $T_{master} = O(|V| + |E|) = (m + l * m)$, where m and l are the number of actions and the maximum expansion factor of each action in DG. Since $m \ll N$ and $l \ll N$ are still satisfiable for a large scale repository, DG can be quickly marked as a choreography master plan.

Example 6 We consider the DG shown in Fig. 6. After using the rules, Fig. 7 illustrates the marking of DG (*talk*'s are not shown). The choreography master plan derived from the marked DG can be written out by our proposed description language P as follows.

ReceiveRFQ; *talk*(S,C); ReceiveQuote; *talk*(C,S); ReceivePO; *talk*(S,W); ReceiveOI; CheckAvail; *talk*(W,S); ReceiveAvail; ($c_1 \triangleright$ (ConfirmPO; (*talk*(S,C); ReceivePOAccept; MakePayment; *talk*(C,S)) ∥ (*talk*(S,W); ReceiveShipOrder; *talk*(W,C); ReceiveShipDetailR; *talk*(C,W); ReceiveShipDetails; *talk*(W,S)); ReceiveShipPayment; *talk*(S,C); ReceiveBuyConfirm)) or ($c_2 \triangleright$ (CancelPO; *talk*(S,C); ReceivePOReject)).

4.6 Generating the distributed choreography plans

To generate distributed plan, we partition DG $= (V, E)$. We partition the vertex set V into multiple, disjoint sets, one for each role $w \in W$. That is, $V = V_1 \cup \cdots \cup V_P$, where $a \in V_i$ if and only if $host(a) = w_i$. For example, the DG in Fig. 6 is partitioned into three vertex sets, according to the role each vertex (action) is associated with. It is illustrated in Fig. 8.

Since $a_i \vdash a_j$ is satisfiable only when $i < j$, the DG is acyclic. Hence, a_j is an **offspring** of a_i if there is a path from a_i to a_j in DG. An action a_i **depends on** a contingency c if a_i is an offspring of b_c in DG.

Definition 14 (Lead Operation) Given a solution plan $\pi = (a_1, \ldots, a_m)$, for a contingency $c \in C$ and service role $w \in W$, the lead operation $lead(c, w)$ is the first action a in π such that $host(a) = w$ and a depends on c. w may not contain a lead operation if no action in w depends on c.

Based on the partitioned DG, for every role, we generate a local plan in the language R defined in Sect. 3. We list the rules for generation below. For each role $w_k \in W, k = 1, \ldots, N$, we consider the actions in partition V_k following the order in π.

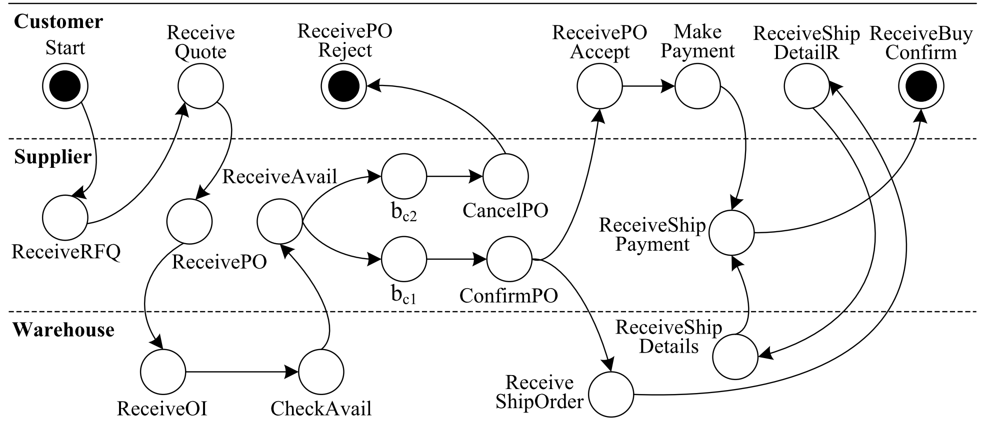


Fig. 8 The partitioning of the DG by service roles

1. For every two actions a_i and a_j in V_k where $i < j$, if a_j is an offspring of a_i, they are arranged sequentially (";"); if not, they are arranged in parallel ("‖") if they depend on the same contingency or by choice ("or") otherwise.
2. For every edge from an operation action in V_k to an action in another partition V_l, insert $send(ch, k, l)$.
3. For every edge from an operation action in a different partition V_l to an action in V_k, insert $recv(ch, l, k)$, where the channel number ch matches the corresponding *send* action.
4. For every contingency action b_c in the partition V_k, insert $send(ch, k, l)$ to every other partition V_l that contains a lead operation.
5. For every action a such that $a = lead(c, w_k)$, where $host(b_c) = l, l \neq k$, insert $recv(ch, l, k)$ before a.

For the generation of choreography distributed plan, the cost is dominated by the vertices partitioning in DG and invocation relationship between two actions in the same partitioning set. Considering the worst case, where the vertices V are divided into only one partitioning set, the time complexity can be calculated by $T_{distribution} = O(|V| + |E| + m * (m-1)/2 * m) = O(m + l * m + m^2 * (m-1)/2)$. As the expansion factor $l < m$ in DG, the time complexity of generating distributed plans is $O(m^3)$. Although it is higher than that of generating a choreography master plan, the distributed plans can still be done fast enough because we only need to make conversion within a very small finite number of actions in DG.

Example 7 Based on the partitioned DG in Fig. 8, we apply the rules above to generate the local plans for three service roles:

Customer: *send*(ch0,C,S); *recv*(ch1,S,C); ReceiveQuote; *send*(ch2,C,S); *recv*(ch5,S,C); ($c_2 \triangleright$ *recv*(ch7,S,C); ReceivePOReject) or ($c_1 \triangleright$ ((*recv*(ch8,S,C); ReceivePOAccept; MakePayment; *send*(ch10,C,S)) ‖ (*recv*(ch11,W,C); ReceiveShipDetailR; *send*(ch12,C,W))); *recv*(ch14,S,C); ReceiveBuyConfirm).

Supplier: *recv*(ch0,C,S); ReceiveRFQ; *send*(ch1,S,C); *recv*(ch2,C,S); ReceivePO; *send*(ch3,S,W); *recv*(ch4,W,S); ReceiveAvail; (*send*(ch5,S,C) ‖ *send*(ch6,S,W)); ($c_2 \triangleright$ CancelPO; *send*(ch7,S,C)) or ($c_1 \triangleright$ ConfirmPO; (*send*(ch8,S,C) ‖ *send*(ch9,S,W)); (*recv*(ch10,C,S) ‖ *recv*(ch13,W,S)); ReceiveShipPayment; *send*(ch14,S,C)).

Warehouse: *recv*(ch3,S,W); ReceiveQI; CheckAvail; *send*(ch4,W,S); *recv*(ch6,S,W); ($c_1 \triangleright$ (*recv*(ch9,S,W); ReceiveShipOrder; *send*(ch11,W,C); *recv*(ch12,C,W); ReceiveShipDetails; *send*(ch13,W,S))).

The correctness of our distributed plan can be validated by showing its equivalence to the master plan, which satisfies the user's need as it solves the planning problem modeling the logical constraints and contingencies of the SCP.

In [28], the authors studied necessary conditions for such equivalence and concluded that a natural projection from a master plan does not always give an equivalent distributed plan. There is a key difference: their work considers any possible master plan, while our approach mainly deal with those master plans and local plans that can be generated from the planning method. Our planning work, although more restrictive, ensures the equivalence of local and master plans by analyzing the dependency relationship and contingency actions and using communication scheme to enforce actions' partial orders.

More specifically, automatic AI planners are applied to find a solution plan on which we perform dependency analysis to ensure the correctness of invocation sequence in a choreography master plan. On the basis of marked dependency graph, partitioning the vertices in DG and decentralization scheme only divide the invocation sequence in the choreography master plan into multiple ones, each for a service role. As a consequence, this kind of partitioning does not make any changes on logically invocation relationships among operations from different service roles.

A counterexample in [28] is a master plan ($a_1^1 \parallel a_1^2$); a_2^1, where a_1^1 and a_2^1 are in role 1 and a_1^2 in role 2. The dis-

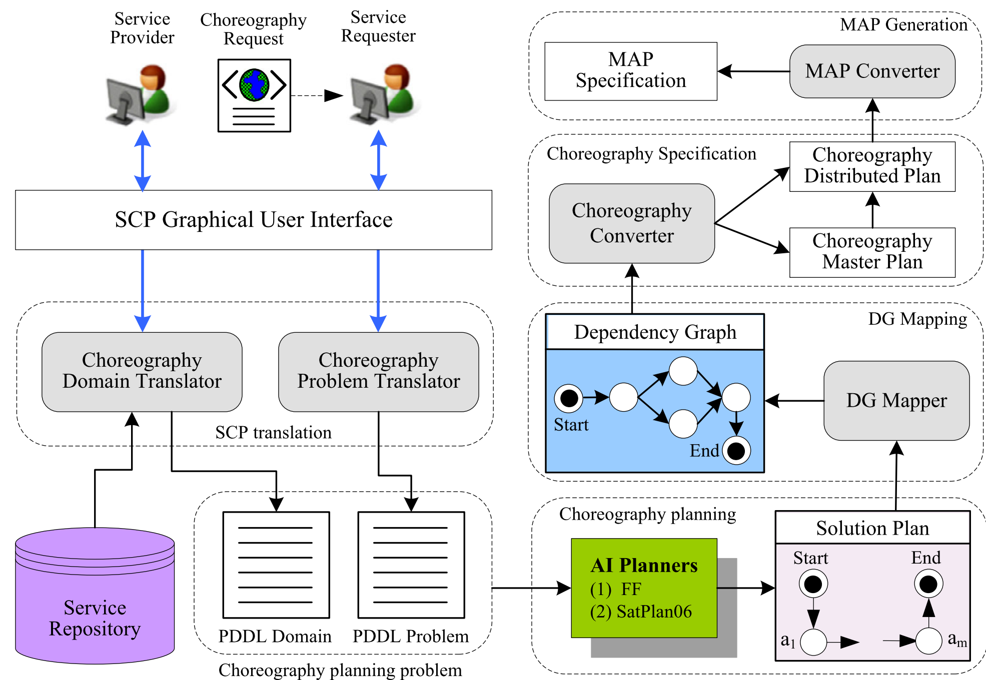


Fig. 9 The system architecture of service choreography

tributed plan is $a_1^1; a_2^1$ for role 1 and a_1^2 for role 2, which allows $a_1^1; a_2^1; a_1^2$, a sequence the master plan does not allow. In our planning approach, however, such a non-equivalence will not occur. If a_2^1 depends on a_1^2, then there will be a send/recv pair that enforces the order; if a_2^1 does not depend on a_1^2, then the choreography master plan will be $(a_1^1; a_2^1) \parallel a_1^2$, which is equivalent to the multiple distributed plans.

From another perspective, under contingencies, it is suggested that equivalence relies on the existence of a *dominant role* [28], which makes choices that all other roles will follow. In our method, we essentially have a dominant role for each contingency (whoever generates the output parameter), and the dominant role sends the decision to the lead operations in other roles.

5 System architecture and implementation

5.1 System architecture

We propose a general framework for automated service choreography using state of the art planners. Figure 9 illustrates the system architecture. There are several system components, including a SCP Graphical User Interface (GUI), a choreography domain translator, a choreography problem translator, a service repository with WSDL description, AI planners (FF [15] and SatPlan06 [18, 19]), a dependency graph (DG) mapper, a choreography specification converter and a MAP converter. In addition to these components, there are two kinds of participants: service provider and service requester. Service providers publish their Web services to the service repository for use. Service requesters consume Web services offered by service providers.

The process of service choreography involves the following steps. First, service providers publish their Web services through GUI to the SCP system, which stores all the registered Web services in the service repository. Second, the choreography domain translator reads all the services from the Web service repository and translates them into a PDDL domain. Third, a service requester submits a choreography request, which is further translated into a PDDL problem by the choreography problem translator. Based on the generated choreography planning problem (consisting of a PDDL domain and a PDDL problem), the SCP system invokes one of the planners (FF [15] or SatPlan06 [18, 19]) to find a solution plan. After that, the SCP system takes the DG mapper to perform dependency analysis between actions on the solution plan to obtain a dependency graph. Then, the choreography converter is used to generate a choreography master plan, which is then converted into choreography distributed plans. Finally, we translate the distributed plans in the R language to the MAP specifications [2]. The MAP translation is described below.

5.2 Translating R to MAP

Based on the distributed choreography plans in the R language, we can directly convert each local plan for a service role into a MAP specification [2]. MAP is a recently proposed executable specification to describe local plans for Web service choreography. A choreography, specified in

Table 1 Distributions of the 18 groups of Web service repositories. Column 'Service Repository' is the service repository name. Column '|W|' is the number of services in a service repository. Column '|P|' is the size of input and output parameters in an operation of a Web service

Composition1			Composition2		
Service Repository	|W|	|P|	Service Repository	|W|	|P|
Composition1-20-4	2156	4–8	Composition2-20-4	3356	4–8
Composition1-20-16	2156	16–20	Composition2-20-16	6712	16–20
Composition1-20-32	2156	32–36	Composition2-20-32	3356	32–36
Composition1-50-4	2656	4–8	Composition2-50-4	5356	4–8
Composition1-50-16	2656	16–20	Composition2-50-16	5356	16–20
Composition1-50-32	2656	32–36	Composition2-50-32	5356	32–36
Composition1-100-4	4156	4–8	Composition2-100-4	8356	4–8
Composition1-100-16	4156	16–20	Composition2-100-16	8356	16–20
Composition1-100-32	4156	32–36	Composition2-100-32	8356	32–36

MAP, can be sent dynamically to a group of distributed peers to execute MAP plans at runtime. For each role $w_k \in W$, $k = 1, \ldots, P$, we use the following rules to implement the conversion from its local plan R_k to the MAP specification.

1. For every two actions a_i and a_j in R_k, if they are sequential (";"), we arrange them sequentially (a_i *then* a_j); if they are parallel, they are arranged in parallel ("a_i *par* a_j").
2. For every two actions a_i and a_j, if they are conditional ("or") and their contingencies are c_i and c_j, respectively, we arrange them by choice ("*if* c_i then a_i *or else if* c_j then a_j").
3. For every send action $send(ch, k, l)$ from w_k to another role w_l, we insert $reply(\$p) => peer(\$w_l)$, where p are communication messages.
4. For every receive action $recv(ch, l, k)$ from w_l to w_k, we insert $request(\$p) <= peer(\$w_l)$.
5. For every receive action $recv(ch, l, k)$ in R_k, we insert a *waitfor* loop before $request(\$p) <= peer(\$w_l)$.
6. For every group of contingencies in R_k, we create a new *method* and use *call* to invoke it.

Based on the above rules, we can convert generated distributed choreography plans to the MAP specifications, where each local plan in R corresponds to a MAP specification.

6 Experimental evaluation

6.1 Experimental setup and datasets

We developed a prototype system in Java. It takes service repositories in WSDL as input, allows a user to specify initial state and choreography goals, and implements our approach to generate a distributed choreography plan in the MAP specification [2]. Two planners FF [15] and SatPlan06 [18, 19] are integrated in our system. We ran our experiments on a PC with Intel Pentium(R) dual core processor 2.4 GHz and 1 G RAM.

We conducted experiments on 81,464 WSDL Web services involved in 18 groups of large scale service repositories. All the datasets are published on ICEBE05[3] and can be freely downloaded from the website of Web Services Challenge. These Web service repositories are categorized into Composition1 and Composition2, which are shown in Table 1.

As shown in the distributions, the number of services involved in a service repository ranges from 2,156 to 8,356, and the size of input or output interface parameters in an operation of a Web service ranges from 4–8, 16–20 to 32–36. In terms of the number of services and parameter size, the easiest dataset to be dealt with is Composition1-20-4. In contrast, the most difficult dataset is Composition2-100-32. Each service repository has 11 choreography requests for use.

Unlike traditional service registration mechanism, the 18 groups of experimentally large scale service repositories are structured by independent service directories. Each corresponds to a finite number of WSDL services as shown in Table 1, although they are distributed into two upper categories by the number of services and interface parameters.

Currently, a service provider needs to specify contingencies when a Web service is published. In fact, this involves only a slight enhancement to a Web service description language such as WSDL or OWL-S. In our experiment, we enhanced WSDL by some special annotations to describe the contingencies. Since only a small number of Web services involve contingencies, it does not require much work as most Web services do not need any changes. Then, the translation from the WSDL repository to the planning formulation is completely automated and does not involve any manual work. That is, our planning translation algorithm will automatically generate the correct planning domain specification in the PDDL language. It can parse the special an-

[3] ICEBE05 provides a set of test data for both service composition and service discovery challenges.

```
1  %customer {
2  method main() =
3    waitfor
4      (request($pid, $pid_name, $pid_quantity) <= peer($enduser)
5      then reply($pid, $pid_name) => peer($supplier)
6      then waitfor
7        (request($pid_price) <= peer($supplier)
8        then ReceiveQuote($pid_price)
9        reply($pid, $pid_quantity) => peer($supplier))
10      timeout(call main())
11     then call wait_decision($pid, $pid_quantity)
12     then call main())
13   timeout(e)
14
15 method wait_decision($pid, $pid_quantity) =
16   waitfor
17     if (request($po_accept) <= peer($supplier))
18       then ((ReceivePOAccept($po_accept)
19       then $payment_info = MakePayment($pid, $pid_quanitity)
20       then reply($payment_info) => peer($supplier))
21       par waitfor
22         (request($shiporder_req) <= peer($warehouse)
23         then $ship_details = ReceiveShipDetailR($shiporder_req)
24         then reply($ship_details) => peer($warehouse))
25       timeout(e))
26       then waitfor
27         (request($purchase_supplier_confirm) <= peer($supplier)
28         then $purchase_confirm = ReceiveBuyConfirm
              ($purchase_supplier_confirm))
29       timeout(e)
30     or else if (request($po_reject) <= peer($supplier))
31       then $po_order_reject = ReceivePOReject($po_reject)
32   timeout(e) }
```

Fig. 10 Local plan of Customer in MAP specification

```
1  %supplier {
2  method main() =
3    waitfor
4      (request($pid, $pid_name) <= peer($customer)
5      then $pid_price = ReceiveRFQ($pid, $pid_name)
6      then reply($pid_price) => peer($customer)
7      then waitfor
8        (reuqest($pid, $pid_quantity) <= peer($customer)
9        then $po_info = ReceivePO($pid, $pid_quantity)
10       then reply($po_info) => peer($warehouse))
11     timeout(call main())
12     then waitfor
13       (request($availability) <= peer($warehouse)
14         then $po_avail = ReceiveAvail($availability))
15     timeout(e)
16     then call order_decision($po_avail)
17     then call main())
18   timeout(e)
19
20 method order_decision($po_avail) =
21   if ($po_avail = avail)
22     then ($po_accept, $shiporderR) = ConfirmPO($po_avail)
23     then (reply($po_accept) => peer($customer)
24     par reply($shiporderR) => peer($warehouse))
25     then waitfor
26       ((request($payment_info) <= peer($customer)
27       par request($ship_confirm) <= peer($warehouse))
28       then $purchase_suplier_confirm = ReceiveShipPayment
            ($payment_info, $ship_confirm)
29       then reply($purchase_supplier_confirm) => peer($customer))
30     timeout(e)
31   or else if ($po_avail = not_avail)
32     then $po_reject = CancelPO($po_avail)
33     then reply($po_reject) => peer($customer) }
```

Fig. 11 Local plan of Supplier in MAP specification

```
1  %warehouse {
2  method main() =
3    waitfor
4      (request($po_info) <= peer($supplier)
5      then $po_check_info = ReceiveOI($po_info)
6      then $availability = CheckAvail($po_check_info)
7      then reply($availability) => peer($supplier)
8      then call wait_shiporder()
9      then call main())
10   timeout(e)
11
12 method wait_shiporder() =
13   waitfor
14     (request($shiporderR) <= peer($supplier)
15     then $shiporder_req = ReceiveShipOrder($shiporderR)
16     then reply($shiporder_req) => peer($customer)
17     then waitfor
18       (request($ship_details) <= peer($customer)
19       then $ship_confirm = ReceiveShipDetails($ship_details)
20       then reply($ship_confirm) => peer($supplier))
21     timeout(e))
22   timeout(e) }
```

Fig. 12 Local plan of Warehouse in MAP specification

notations for contingencies and correctly translate them into contingency actions.

Taking the choreography dependency graph in Fig. 8 as an example, we converted it into three MAP specifications. Figures 10, 11, and 12 show the local plans for Customer, Supplier and Warehouse, respectively. The translation from our language *R* to MAP is direct from the example. Details of MAP specifications are described in [2]. Starting from a product order request, the three MAP specifications as three roles collaborate on a product purchase task.

Since the most time-consuming components in choreography are SCP translation and plan generation, we report the cost of these two parts, respectively.

6.2 SCP translation

We first tested the SCP translation performance of our approach on 81,464 Web services involved in the 18 groups of service repositories. Table 2 presents the time cost of SCP domain translation and average SCP problem translation. The results indicate that the average time for generating a PDDL problem is short, ranging from 1.36 milliseconds to 14.27 milliseconds. However, with the increasing number of services involved in different service repositories, the SCP domain translation time takes from 74.031 seconds to 1,103.38 seconds.

Although the most difficult Web service repository with the largest number of services and I/O parameters (Composition2-100-32) takes more than 1,100 seconds for the SCP translation, it can be performed offline only once. To find a service choreography, the user only needs to specify the inputs and possible choreography goals. The system will automatically generate the adjusted planning formulation very quickly. In other words, once we translate operations in a large Web service repository into planning actions, the service repository does not need to be parsed again when a user submits a choreography task.

6.3 Response time of finding a solution plan

From the view of practicality in real world applications, the response time is of vital importance, because it determines

Table 2 SCP Translation time for generating a choreography PDDL domain and a choreography PDDL problem on all of the 18 groups of service repositories. Column 'SCP Dom' is the SCP domain translation time for a service repository. Column 'SCP Prob' is the average SCP problem translation time for all 11 choreography requests of a service repository

Composition1			Composition2		
Service Repository	SCP Dom	SCP Prob	Service Repository	SCP Dom	SCP Prob
Composition1-20-4	74.031 s	1.36 ms	Composition2-20-4	164.922 s	1.36 ms
Composition1-50-4	109.765 s	1.45 ms	Composition2-50-4	400.140 s	2.91 ms
Composition1-100-4	250.922 s	1.45 ms	Composition2-100-4	964.031 s	2.82 ms
Composition1-20-16	75.641 s	1.45 ms	Composition2-20-16	625.453 s	2.82 ms
Composition1-50-16	112.250 s	2.91 ms	Composition2-50-16	412.312 s	2.82 ms
Composition1-100-16	253.000 s	2.91 ms	Composition2-100-16	972.469 s	1.45 ms
Composition1-20-32	77.203 s	2.82 ms	Composition2-20-32	175.735 s	8.55 ms
Composition1-50-32	112.953 s	2.82 ms	Composition2-50-32	417.438 s	11.18 ms
Composition1-100-32	260.172 s	14.27 ms	Composition2-100-32	1,103.38 s	8.45 ms

whether a choreography solution plan can be rapidly returned to the users within a short period of time. Therefore, we adopt response time as the evaluation metric to compare the performance of our approach with WSPR [23, 24] throughout our experiments.

We compared the response time of finding a choreography solution plan with WSPR [23, 24], since it is a well-known solver for automated composition of Web services using AI planning techniques. It applies forward and regression search algorithm to find a solution plan. WSPR solves service orchestration, a much simpler problem than the choreography problem that our approach solves, although we compare the response time of finding a solution plan between these two approaches for dynamic composition of Web services in large scale service repositories.

The response time lasts the duration from the users submitting a service request until receiving a solution plan or failing to find a solution. Specifically, the response time in our approach lasts the duration, including translating a choreography request r to its choreography PDDL problem $\mathcal{P}$, parsing PDDL problem $\mathcal{P}$ and PDDL domain $\mathcal{D}$, and applying a planner to find a solution plan π. On the other hand, WSPR takes its response time by parsing a request r, parsing services in a service repository W, and searching for a solution plan π using forward and regression search. Here, we compared the efficiency of our planning approach with that of WSPR, although WSPR solves a simpler composition problem without contingencies while our planners solve the more complex choreography problem.

In order to validate the efficiency of our approach in finding a solution plan, we tested our approach and WSPR on all the 18 groups of service repositories containing 81,464 services from the ICEBE05 Web services composition challenge. Each service repository has 11 choreography requests for use. The response time (RT) to all 11 choreography requests on each service repository is illustrated in Fig. 13 by our approach using FF [15] and SatPlan06 [19] and WSPR. Moreover, Table 3 summarizes the average response time for all 11 choreography requests on each service repository of our service choreography planning approach and WSPR.

With regard to the service number, all of them take longer to find a solution plan when the number of parameters becomes larger in a service repository. However, the response time using FF or SatPlan06 increases substantially slower than WSPR. More specifically, for the largest instance, FF takes 14.833 seconds, SatPlan06 takes 92.536 seconds, while WSPR takes 148.807 seconds. The results show that our approach using FF and SatPlan06 is faster and has better scalability than WSPR. Moreover, our approach can automatically find distributed plans, while WSPR and other existing work can only find centralized composition plans.

Comparing to WSPR, we can see that our approach solves more complex choreography problem and handles distributed collaboration, communication and contingency. However, it is more efficient than WSPR in finding a solution plan in regard to the average response time. The reason is that our translation to PDDL allows us to leverage on the advances of automated planners, while WSPR uses its own planning model and composition solver.

From the experimental results of response time on each service repository, we conclude that our approach using FF will most likely lead to better performance for a SCP to find a solution plan with a short period of response time and good scalability, so that we can quickly project it into a choreography dependency graph and MAP specifications.

6.4 Discussion

During the generation of distributed plans for service choreography specification, all the datasets are from WS challenge benchmark for composition performance evaluation. Web services organized in different 18 groups of repositories are disjoint in terms of functional capabilities. In other words, no redundant services exist in a translated choreography planning domain, when finding a solution plan with

Table 3 The average response time in seconds of finding a choreography solution plan of our approach using FF and SatPlan06 planners and WSPR. There are 18 service repositories on ICEBE05. Each repository has 11 choreography requests for use. Column '|P|' is the size of input and output parameters in an operation of a Web service. Column '|W|' is the number of services in a service repository. Column 'Generation time' is the average response time of finding a solution plan for all 11 choreography requests on a service repository

Service repository	|P|	|W|	Generation time		
			FF	SatPlan06	WSPR
Composition1-20-4	4–8	2156	0.306	0.800	8.674
Composition1-50-4		2656	0.344	0.980	11.242
Composition1-100-4		4156	0.491	1.539	17.665
Composition1-20-16	16–20	2156	1.149	2.334	17.753
Composition1-50-16		2656	1.198	3.565	22.478
Composition1-100-16		4156	1.873	5.419	36.278
Composition1-20-32	32–36	2156	3.224	5.076	29.988
Composition1-50-32		2656	3.422	7.910	37.726
Composition1-100-32		4156	4.792	14.221	62.629
Composition2-20-4	4–8	3356	0.569	2.672	14.878
Composition2-50-4		5356	0.845	4.391	24.046
Composition2-100-4		8356	1.214	7.373	38.934
Composition2-20-16	16–20	6712	4.359	30.210	63.430
Composition2-50-16		5356	3.817	18.307	49.776
Composition2-100-16		8356	5.320	32.235	81.791
Composition2-20-32	32–36	3356	7.819	26.609	50.719
Composition2-50-32		5356	11.098	49.489	86.794
Composition2-100-32		8356	14.833	92.536	148.807

the adoption of efficient automatic planners. Fortunately, our approach can also adapt to the scenarios where there are redundant services, because high level planning techniques that apply heuristic search with logical reasoning can ensure the correctness of finding a sequence of actions, so that the found plan transforms the initial state to a state that satisfies the goal specification.

However, we cannot handle the problem where violations of one of the actions in a choreography distributed plan occur with unexpected outputs at execution phase. It depends on the concrete selection of a specific service at runtime by a service execution monitor. Instead of the investigation on service choreography at execution level, we mainly focus on finding a solution plan which is converted into a choreography master plan and multiple distributed plans at design and planning level. One possibility for our extended work to solve this problem is that, AI techniques such as replanning may be exploited to dynamically select a new service to replace the one that does not work with expectation for service choreography at execution level.

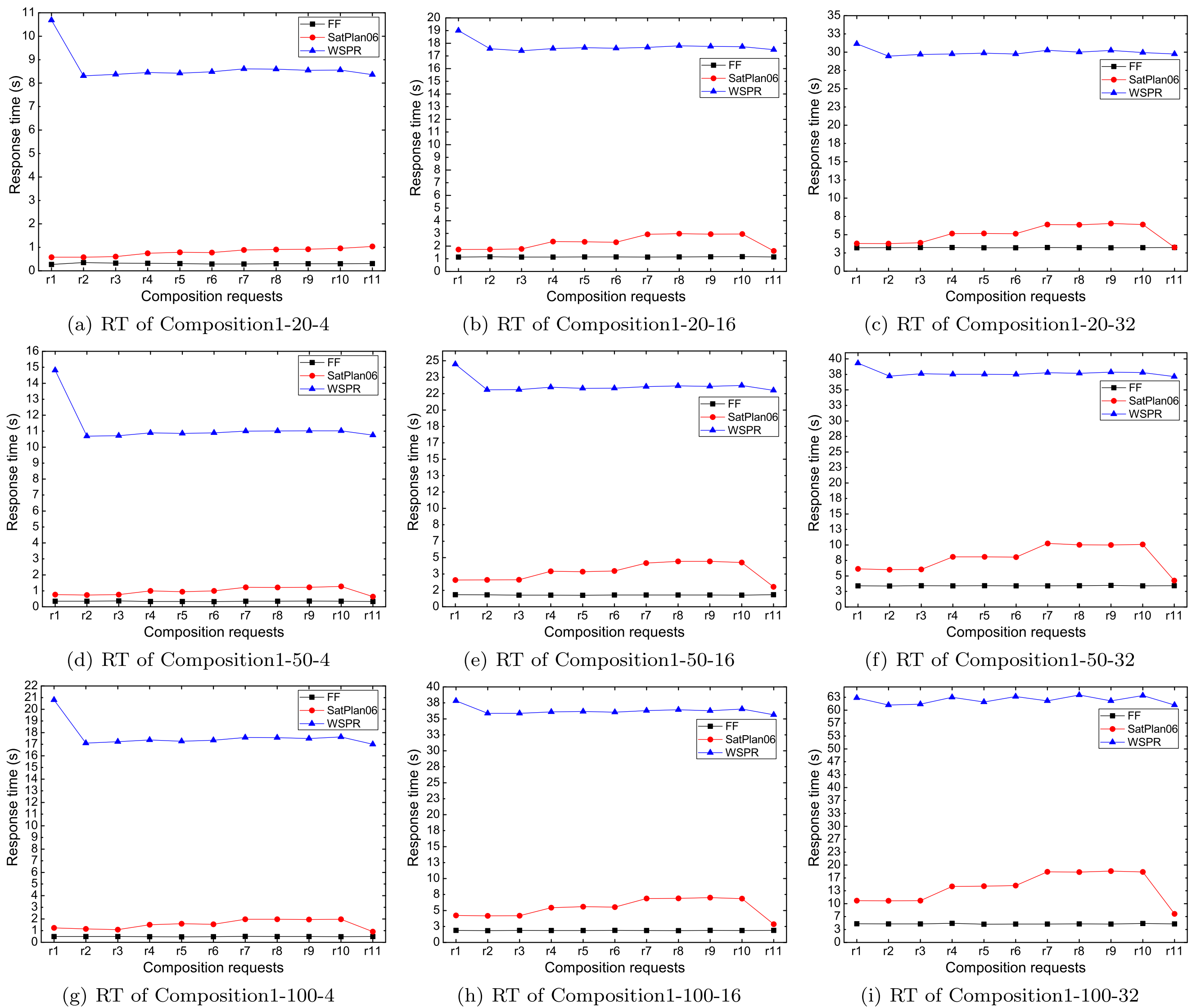
For the contribution of our work, we compile a SCP as a choreography planning problem with the availability of multiple contingencies. Accordingly, we translate a Web service composition problem considering uncertainty into a deterministic planning problem, which can be solved with the exploitation of state of the art automatic planners. Although two highly efficient planners have been adopted to find a solution plan, the major concentration of this work on the basis of a solution plan lies in the automated choreography of Web services, which involves dependency analysis, dependency graph analysis, decentralization scheme, and communication control. First, dependency analysis on a solution plan is performed to construct a dependency graph, which mirrors the invocation relationships between two operations of Web services. Moreover, dependency graph analysis and decentralization scheme have been proposed to generate a choreography master plan and multiple distributed plans.

Currently, our approach is compatible with three kinds of invocation relationships, including sequential, parallel and conditional. However, there are still other control flows such as the support of loop execution in practical applications that we have not taken into account in this work. It could be further investigated by the extension of our existing dependency graph analysis and decentralization scheme.

7 Related work

Our work is related to WSC and AI planning. We classify WSC approaches by planning from four aspects.

– **Orchestration or choreography**. There are two ways to describe Web service composition, i.e., service orchestration and service choreography [25]. Service orchestration has a central controller to coordinate all of the participating services. There are approaches [3, 27] that study service orchestration using BPEL4WS specification to de-



(a) RT of Composition1-20-4 (b) RT of Composition1-20-16 (c) RT of Composition1-20-32

(d) RT of Composition1-50-4 (e) RT of Composition1-50-16 (f) RT of Composition1-50-32

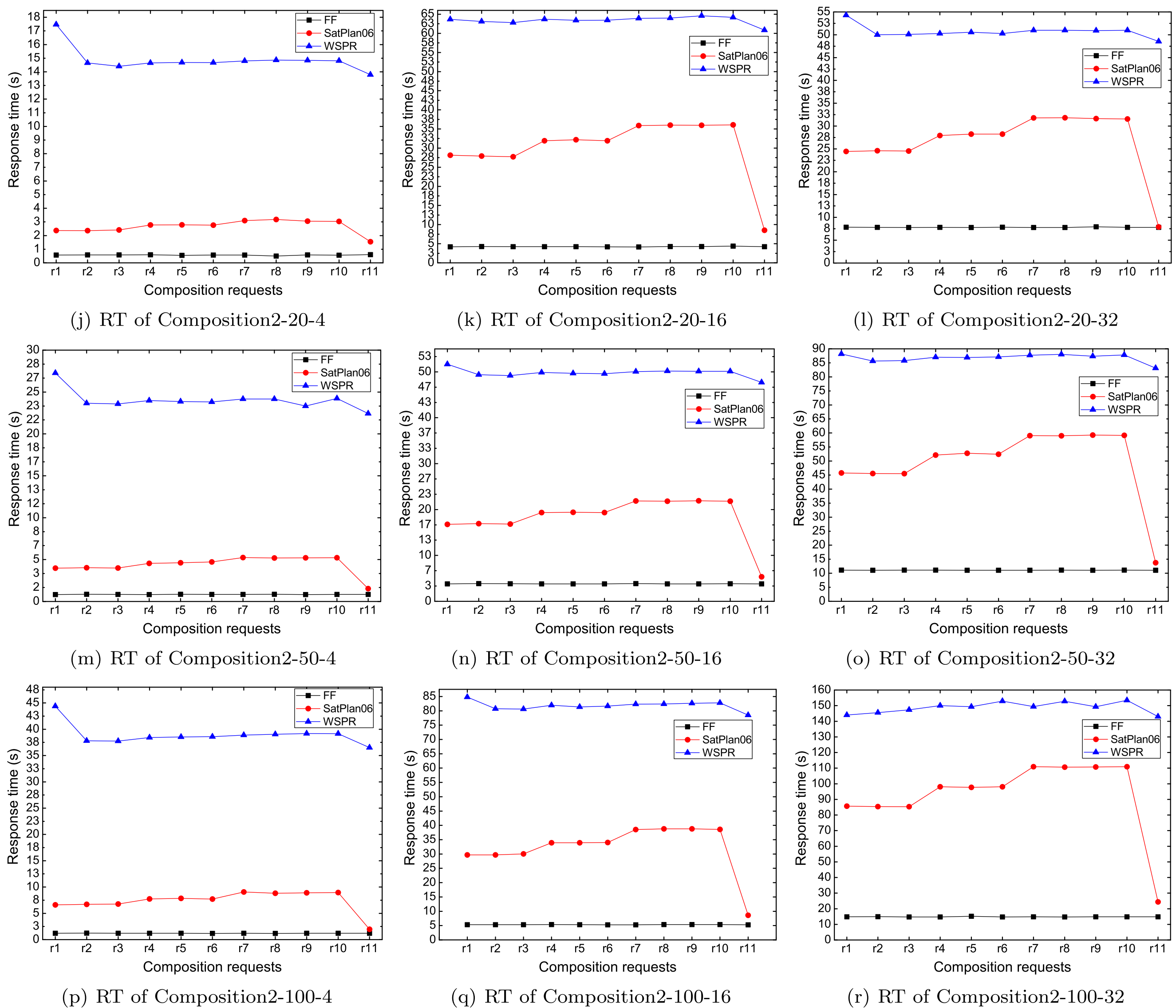
(g) RT of Composition1-100-4 (h) RT of Composition1-100-16 (i) RT of Composition1-100-32

Fig. 13 The response time (RT) of each 11 composition requests on their corresponding dataset in 18 groups of Web service repositories among three WSC approaches FF, SatPlan06 and WSPR

scribe executable business processes. Differently, service choreography tries to achieve a globally shared task by collaboration of multiple services. It has received many attentions due to its multiple key advantages over service orchestration, such as less data transfer and robustness. Some research efforts have been made for service choreography [2, 17, 28, 30]. However, they need to manually generate a Web service choreography specification.

– **AI search or planner**. For automatic composition of Web services, some approaches use AI search techniques to find a composition solution, such as heuristic forward and regression search [23] and planning graph construction [32]. On the contrary, some other approaches [8, 13, 20, 29] directly use AI planner to find a solution plan. In particular, to solve the possible real world WSC problems, including partial observability of the environment, nondeterministic effects and execution failures of Web services, a novel AI planner [21] called Simplanner is designed and implemented for working in excessively dynamic environments.

– **Determinism or uncertainty**. Many composition approaches are deterministic, where the initial states and action outcomes are deterministic. There are also some approaches that take the uncertainty of the initial state into consideration, such as [10, 12]. In general, these approaches transform a WSC problem to a conformant planning problem with multiple possible initial states. The advantage is that they allow users to specify a composition



(j) RT of Composition2-20-4

(k) RT of Composition2-20-16

(l) RT of Composition2-20-32

(m) RT of Composition2-50-4

(n) RT of Composition2-50-16

(o) RT of Composition2-50-32

(p) RT of Composition2-100-4

(q) RT of Composition2-100-16

(r) RT of Composition2-100-32

Fig. 13 (*Continued*)

request with uncertain initial conditions. However, solving a conformant planning problem is a time-consuming task due to its belief update [14]. Moreover, a solution to a conformant planning problem must fulfill each state in the final belief which results in more actions and incurs expensive communication cost.

– **Semantic or syntactic representation**. In terms of Web service representation, we can divide WSC approaches into semantic and syntactic ones. The semantic approaches [13, 20] can handle semantic match between operations where Web services are represented by a Web service ontology model such as OWL-S or DAML-S. Nonetheless, the semantic approach still faces many challenges in practical applications, because most available services published on the Web are described in WSDL. Some other WSC methods [23, 32] are based on syntactic representations.

In addition to the application of AI planning techniques or off-the-shelf highly efficient automated planners, agent-based techniques have also been exploited for solving WSC problems, such as composing services in multi-cloud environments for different kinds of cloud services [11] and providing business services to conquer the crisis and enhance autonomic service cooperation [9]. Furthermore, to improve the quality of service selection [31] for dynamic composition of Web services, Learning Automata (LA) solution which has proven to be capable of learning the optimal action has been used to efficiently identify high quality Web services when operating in unknown stochastic environments.

8 Conclusions and future work

Automatic and efficient Web service composition can simplify the implementation of business processes. Service choreography is an important paradigm for WSC with many advantages over service orchestration. This paper presents an effective and efficient planning-based method for automated generation of Web service choreography and proposes a number of novel techniques, including compilation of contingencies, dependency analysis, dependency graph analysis, and communication control among actions.

The method first models a SCP as a classical planning problem and solves it using state of the art planners. Then, the method performs dependency analysis among actions on the solution plan, and constructs a choreography dependency graph to express the invocation order among different operation actions and contingency actions. Finally, the method automatically generates a master plan and distributed plans for Web service choreography based on dependency graph analysis. We have conducted extensive experiments on large scale Web service repositories. The experimental results show that the proposed method is effective and efficient and can be fully automated. Comparison to an existing planning-based orchestration approach shows that our approach has superior scalability, although it solves the more complex choreography problems.

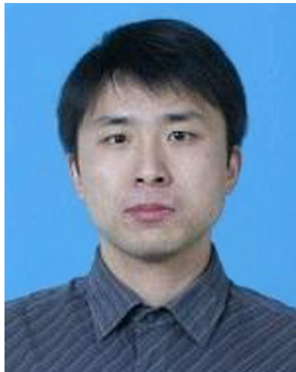
Our future work includes two directions: extending our current approach to support global constraints linking multiple users for service choreography, and optimization of plan quality by taking into account important nonfunctional metrics, such as the QoS of Web services.

Acknowledgements We thank Jörg Hoffmann, Henry Kautz and Bart Selman for providing open sources of AI planners FF and SatPlan06. We appreciate all of the three anonymous reviewers for insightful comments.

References

1. Agarwal V, Chafle G, Dasgupta K et al (2005) Synthy: a system for end to end composition of Web services. J Web Semant 3(4):311–339
2. Barker A, Walton CD, Robertson D (2009) Choreographing Web services. IEEE Trans Serv Comput 2(2):152–166
3. Bertoli P, Kazhamiakin R, Paolucci M et al (2009) Continuous orchestration of Web services via planning. In: Proceedings of the international conference on automated planning and scheduling (ICAPS)
4. Bertoli P, Pistore M, Traverso P (2010) Automated composition of Web services via planning in asynchronous domains. Artif Intell 174(3):316–361
5. Busi N, Gorrieri R, Guidi C et al (2006) Choreography and orchestration conformance for system design. In: Proceedings of the international conference on coordination models and languages (COORDINATION)
6. Chen L, Wassermann B, Emmerich W et al (2006) Web service orchestration with BPEL. In: Proceedings of the international conference on software engineering (ICSE)
7. Daniel F, Pernici B (2006) Insights into Web service orchestration and choreography. Int J E-Bus Res 2(1):58–77
8. Falou ME, Bouzid M, Mouaddib AI et al (2010) A distributed planning approach for Web services composition. In: Proceedings of the IEEE international conference on Web services (ICWS)
9. Gao J, Lv H (2012) Institution-governed cross-domain agent service cooperation: a model for trusted and autonomic service cooperation. Appl Intell 37(2):223–238
10. Giacomo GD, Masellis RD, Patrizi F (2009) Composition of partially observable services exporting their behaviour. In: Proceedings of the international conference on automated planning and scheduling (ICAPS)
11. Gutierrez-Garcia JO, Sim KM (2013) Agent-based cloud service composition. Appl Intell 38(3):1–29
12. Hoffmann J, Bertoli P, Helmert M et al (2009) Message-based Web service composition, integrity constraints, and planning under uncertainty: a new connection. J Artif Intell Res 35(1):49–117
13. Hoffmann J, Bertoli P, Pistore M (2007) Web service composition as planning, revisited: in between background theories and initial state uncertainty. In: Proceedings of the national conference on artificial intelligence (AAAI)
14. Hoffmann J, Brafman RI (2006) Conformant planning via heuristic forward search: a new approach. Artif Intell 170(6–7):507–541
15. Hoffmann J, Nebel B (2001) The FF planning system: fast plan generation through heuristic search. J Artif Intell Res 14(1):253–302
16. Hwang SY, Lim EP, Lee CH et al (2008) Dynamic Web service selection for reliable Web service composition. IEEE Trans Serv Comput 1(2):104–116
17. Kang Z, Wang H, Hung P (2007) WS-CDL+: an extended WS-CDL execution engine for Web service collaboration. In: Proceedings of the IEEE international conference on Web services (ICWS)
18. Kautz H, Selman B (1999) Unifying SAT-based and graph-based planning. In: Proceedings of the international joint conference on artificial intelligence (IJCAI)
19. Kautz H, Selman B, Hoffmann J (2006) SatPlan: planning as satisfiability. In: Abstracts of the international planning competition (IPC)
20. Klusch M, Gerber A, Schmidt M (2005) Semantic Web service composition planning with OWLS-XPlan. In: Proceedings of the AAAI fall symposium on semantic Web and agents
21. Kuzu M, Cicekli NK (2012) Dynamic planning approach to automated Web service composition. Appl Intell 36(1):1–28
22. Meng S, Arbab F (2007) Web services choreography and orchestration in Reo and constraint automata. In: Proceedings of the 2007 ACM symposium on applied computing (SAC)
23. Oh SC, Lee D, Kumara SRT (2007) Web service planner (WSPR): an effective and scalable Web service composition algorithm. Int J Web Serv Res 4(1):1–22
24. Oh SC, Lee D, Kumara SRT (2008) Effective Web service composition in diverse and large-scale service networks. IEEE Trans Serv Comput 1(1):15–32
25. Peltz C (2003) Web services orchestration and choreography. Computer 36(10):46–52
26. Pistore M, Marconi A, Bertoli P et al (2005) Automated composition of Web services by planning at the knowledge level. In: Proceedings of the international joint conference on artificial intelligence (IJCAI)
27. Pistore M, Traverso P, Bertoli P (2005) Automated composition of Web services by planning in asynchronous domains. In: Proceedings of the international conference on automated planning and scheduling (ICAPS)

28. Qiu Z, Zhao X, Cai C, Yang H (2007) Towards the theoretical foundation of choreography. In: Proceedings of the international World Wide Web conference (WWW)
29. Sirin E, Parsia B, Wu D et al (2004) HTN planning for Web service composition using SHOP2. J Web Semant 1(4):377–396
30. Yang H, Zhao X, Cai C, Qiu Z (2008) Model-checking of Web services choreography. In: Proceedings of the IEEE international symposium on service-oriented system engineering
31. Yazidi A, Granmo OC, Oommen BJ (2012) Service selection in stochastic environments: a learning-automaton based solution. Appl Intell 36(3):617–637
32. Zheng XR, Yan YH (2008) An efficient syntactic Web service composition algorithm based on the planning graph model. In: Proceedings of the IEEE international conference on Web services (ICWS)
33. Zou G, Chen Y, Xu Y et al (2012) Towards automated choreographing of Web services using planning. In: Proceedings of the national conference on artificial intelligence (AAAI)

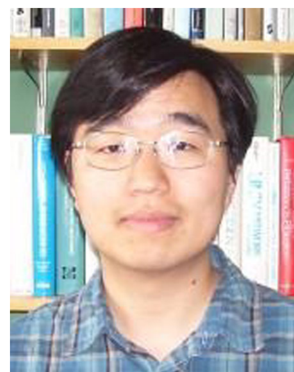


Guobing Zou is an assistant professor in the School of Computer Engineering and Science at Shanghai University, China. He received a Ph.D. degree in computer science from Tongji University, Shanghai, China, 2012. He has worked as a visiting scholar in the Department of Computer Science and Engineering at Washington University in St. Louis from 2009 to 2011, USA. His current research interests mainly focus on Web service composition, service discovery and uncertain planning. He has published more than 20 papers on international journals and conferences, including IEEE Transactions on Services Computing (TSC), AAAI, Soft Computing and CCV. He served as a program committee member on CIT-12, UUMA-12 and UUMA-13, and organization member on DISD-13. He worked as a reviewer for Journal of Artificial Intelligence Research (JAIR), IEEE Transactions on Services Computing (TSC), KDD-09, AAAI-10, ICDM-10, IJCAI-11 and KDD-11.

Yanglan Gan is an assistant professor in the school of computer science and technology at Donghua University, Shanghai, China. She received her Ph.D. degree in computer science from Tongji University in 2012. Her research interests include Bioinformatics, data mining, Web services, and information retrieval. She has published more than 15 papers on international journals and conferences, including Bioinformatics, BMC Bioinformatics, Knowledge-Based Systems, Soft Computing and FSKD. She served as a program committee member on APBC-12, ADMA-13 and APBC-14. She worked as a reviewer for varieties of international journals and conferences, such as BMC Bioinformatics, Knowledge-Based Systems, KDD-10, IJCAI-11, APBC-12, ADMA-13 and APBC-14.

Yixin Chen is an associate professor of computer science at the Washington University in St. Louis. He received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 2005. His research interests include nonlinear optimization, constrained search, planning and scheduling, data mining, and data warehousing. His work on planning has won First-Class Prizes in the International Planning Competitions (2004 and 2006). He has won the Best Paper Award in AAAI (2010) and ICTAI (2005), and Best Paper nomination at KDD (2009). He has received an Early Career Principal Investigator Award from the Department of Energy (2006) and a Microsoft Research New Faculty Fellowship (2007). Dr. Chen is a senior member of IEEE. He serves as an associate editor on the IEEE Transactions on Knowledge and Data Engineering, and ACM Transactions on Intelligent Systems and Technology.

Bofeng Zhang is a full professor in the School of Computer Engineering and Science at Shanghai University. He received his Ph.D. degree from the Northwestern Polytechnic University (NPU) in 1997, China. He experienced a Postdoctoral Research at Zhejiang University from 1997 to 1999, China. He worked as a visiting professor at the University of Aizu from 2006 to 2007, Japan. His research interests include personalized service recommendation, intelligent human-computer interaction, and data mining. He has published more than 120 papers on international journals and conferences. He worked as the program chair for UUMA-11, UUMA-12 and UUMA-13. He also served as a program committee member for lots of international conferences.



Ruoyun Huang is currently a software engineer at Google, working in large scale intelligent systems. He received the Ph.D degree in computer science from Washington University in St. Louis in August, 2011. His research interests include automated planning, large scale intelligent systems, Web service composition, and probabilistic inference. He has published more than 12 papers on top journals and conferences, including Journal of Artificial Intelligence Research (JAIR-12), Artificial Intelligence (AIJ-09), AAAI (2008, 2010, 2012) and ICAPS-09. He won AAAI'10 Outstanding Paper Award. Before his Ph.D study, he also worked in Bearingpoint consulting.

You Xu is currently a Ph.D. candidate in the department of computer science and engineering at Washington University in St. Louis. He received the B.Sc. degree in mathematics from Nanjing University in 2006, and the M.Sc. degree in computer science from Washington University in St. Louis in 2009. His research interests include large-scale nonlinear optimization, constrained search, and partial order reduction for planning, and automated planning in cloud computing. He has published 12 papers, including RTAS-12, RTAS-11, MobiHoc-10, RTSS-10, and IJCAI-09.

Yang Xiang is a professor in the department of computer science and technology at Tongji University. He received the Ph.D. degree in management science and engineering from Harbin Institute of Technology in 1999, China. His research interests include data warehousing and data mining, intelligent decision support system, service computing and e-commerce. He has published more than 100 papers on the international journals and conferences, including Expert Systems with Applications, Science China Information Sciences, and Chinese Journal of Electronics. He has published 4 books on intelligent decision support system.