Towards Automated Choreographing of Web Services Using Planning

Guobing Zou†‡§, Yixin Chen‡, You Xu‡, Ruoyun Huang‡, Yang Xiang§
†School of Computer Engineering and Science, Shanghai University, China
‡Dept. of Computer Science and Engineering, Washington University in St. Louis, USA
§Dept. of Computer Science and Technology, Tongji University, China
guobingzou@gmail.com, {chen, yx2, rh11}@cse.wustl.edu, shxiangyang@tongji.edu.cn

Abstract

For Web service composition, choreography has recently received great attention and demonstrated a few key advantages over orchestration such as distributed control, fairness, data efficiency, and scalability. Automated design of choreography plans, especially distributed plans for multiple roles, is more complex and has not been studied before. Existing work requires manual generation assisted by model checking. In this paper, we propose a novel planning-based approach that can automatically convert a given composition task to a distributed choreography specification. Although planning has been used for orchestration, it is difficult to use planning for choreography, as it involves decentralized control, concurrent workflows, and contingency. We propose a few novel techniques, including compilation of contingencies, dependency graph analysis, and communication control, to handle these characteristics using planning. We theoretically show the correctness of this approach and empirically evaluate its practicability.

Introduction

Orchestration and choreography are two ways for Web service composition. A WSC problem can be described from the view of a single participant using orchestration or from a global perspective using choreography (Barker, Walton, and Robertson 2009). Service orchestration refers to an executable business process that has a central controller process to coordinate all of the participating Web services (Peltz 2003). On the contrary, service choreography does not have an orchestrator, but all of the participating Web services collaborate with each other in order to achieve a shared goal. The Web Services Choreography Description Language (WS-CDL) is an XML-based W3C candidate language for describing collaborations of service choreography.

It has been widely recognized that choreography has certain key advantages over orchestration, including less data transfer, robustness under server failure, fairness, and avoidance of deadlocks. Although service choreography is widely advocated, automated generation of service choreography specification has not been studied. WS-CDL describes choreography from a global view in a single master plan. It is realized that such a global view is not sufficient and choreography plans should be distributed (Qiu et al. 2007). Although a global view such as defined by WS-CDL is helpful, at the execution end, each role should have a "local plan" such as the MAP protocol (Barker, Walton, and Robertson 2009) that specifies what it needs to do from its individual perspective. Designers ensure that the collaboration of the multiple roles correctly achieves the global goals.

Some recent research focuses on new languages for distributed plans in choreography, but relies on users to manually write the specifications. Examples include the Multiagent Protocols (MAP) (Barker, Walton, and Robertson 2009), and WS-CDL+ (Kang, Wang, and Hung 2007). They do not solve the problem of how to generate these specifications, although there are works that verify the protocol using model checking (Barker, Walton, and Robertson 2009; Yang et al. 2008). Due to the complexity of decentralized logics, manually completing such a specification for service choreography can be tedious and error prone. Although model checking can be used to verify the plan, designing by trial-and-error is labor intensive.

In this paper, we propose a novel approach that can automatically generates a distributed choreography plan based on automated planning. Although planning has been used for orchestration, it is difficult for choreography for several challenges. While planning is suitable for constructing the composition plan from the view of a single party in orchestration, choreography needs distributed and coordinated plans for multiple participants, making planning more difficult. Also, choreography needs to support asynchronous communication between peers and contingent plans that depend on the outcomes of Web services.

We propose a novel planning-based framework to address the above challenges. Our framework compiles a repository of related Web services, along with user-defined contingencies, into a planning domain in Planning Domain Definition Language (PDDL). This process compiles away the contingency on action outcomes into a deterministic planning problem. A planner is used to find a master plan which provides a global view of the choreography. We further propose a decentralization scheme which uses a dependency analysis to support the synthesis of local plans, one for each peer. By addressing three main features in choreography, includ-
The theory and principles of automated choreography by planning, and extensive experiments on 81,464 WSDL services in 18 groups of large scale service repositories. The experimental results validated the feasibility and efficiency of our proposed work.

A Running Example

Our example has three Web services: customer, supplier and warehouse. Figure 1 shows an ordering process.

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 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 the Supplier receives it (ReceiveAvail) from the Warehouse, it makes a decision for the order request.

There are two possibilities. If a product is unavailable, the Supplier cancels it (CancelPO) and notifies the Customer. Otherwise, the Supplier accepts it (ConfirmPO). After receiving the acceptance (ReceivePOAccept), the Customer makes a payment (MakePayment) and replies a payment confirmation to the Supplier. Meanwhile, after receiving a shipping order request (ReceiveShipOrder), the Warehouse sends a request to the Customer for shipping details. Once the Customer receives it (ReceiveShipDetailR), it replies shipping details to the Warehouse. After receiving a payment and shipping confirmation (ReceiveShipPayment), the Supplier replies to a purchase confirmation to Customer.

Problem Formulation

We first formulate our problem using a simplified model. In this paper, we focus on understanding the fundamental principles of automated choreography by planning, and simplify or omit certain issues in Web service composition, such as ontology (Agarwal et al. 2005) and background theory (Hoffmann, Bertoli, and Pistore 2007). The theory and algorithms can be readily incorporated with other techniques and extended to more complex systems. In the experimental results section, we present an evaluation using real-world WSDL Web services and MAP for choreography description. As we will see, the projection from our theoretical model to these languages is straightforward.

Definition 1 (Web Service). A Web service w consists of a set of operations, denoted as \( w = \{op_1, op_2, \cdots \} \), where \( \forall op \in w \) is a 2-tuple \( (I, O) \), where \( I = \{I^1, I^2, \cdots \} \) is a set of input interface parameters. Similarly, \( O = \{O^1, O^2, \cdots \} \) 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 \( \text{Dom}(x) \) to denote its possible values and \( x.value \in \text{Dom}(x) \) 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). For an operation \( op \) with \( op.I = \{I^1, I^2, \cdots \} \), a contingency is a tuple \( c = (op, I^1, pre^1) \) where \( pre^1 \subseteq \text{Dom}(I^1) \).

A contingency \( c = (op, I^1, pre^1) \) means that to invoke \( op \), we need its input parameter \( I^1 \) to take the values in \( pre^1 \), instead of any value in \( \text{Dom}(I^1) \).

We define a service state as a set of interface parameters \( Q = \{x^1, x^2, \cdots \} \) that are defined and assigned values. 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 \( Q \triangleright op \). An operation \( op \) under contingency \( c = (op, I^1, pre^1) \) is applicable at \( Q \) if \( op.I \subseteq Q \) and \( op.I.value \in pre^1 \). We denote this as \( Q \triangleright c \triangleright op \).

When an applicable operation \( op \) or \( c \triangleright op \) is applied to \( Q \), the resulting state \( Q' = Q \oplus op \) (or \( Q' = Q \oplus c \triangleright op \)) is \( Q' = Q \cup op.O \), in which the values of the parameters in \( op.O \) are set by executing \( op \). An execution sequence is an ordered list \( L = (o_1, \cdots, o_m) \) where each element is either an operation \( op \) or an operation with contingency \( c \triangleright op \). Applying a sequence \( L \) to a service state \( Q \) results in \( Q'' = Q \odot L = (((Q \oplus o_1) \oplus o_2) \cdots \oplus o_m) \) if every step is applicable (otherwise \( Q \odot L \) is undefined).

Definition 4 (Service Choreography Problem (SCP)). A SCP is defined by \( (W, C, r_{in}, r_{out}) \): 1) a service repository \( W = \{w_1, \cdots, w_N\} \), 2) a set of contingencies \( C = \{c_1, \cdots, c_N\} \), 3) an input parameter set \( r_{in} = \{q^1, q^2, \cdots\} \), and 4) an output parameter set \( r_{out} = \{q^1, q^2, \cdots\} \).

Example 1. Following our running example, its SCP has \( W = \{\text{Customer, Supplier, Warehouse}\} \). An example operation is ReceiveRFQ \( \{I, O\} \), where \( I = \{\text{pid, pid_name}\}, O = \{\text{price}\} \). Another operation is ConfirmPO \( \{I, O\} \), where \( I = \{\text{po_avail}\}, O = \{\text{po_accept, shiporderR}\} \). In particular, \( \text{Dom}(po_avail) = \{\text{avail, not_avail}\} \), while \text{po_accept} and \text{shiporderR} represent an order acceptance and shipping order request, respectively.

The contingency set consists of \( C = \{c_1, c_2\} \), where \( c_1 = \{\text{ConfirmPO, po_avail}, \{\text{avail}\}\} \) and
We develop an approach that can correctly generate a distributed solution for a SCP. It has a few major steps. 1) Translate an SCP into a PDDL planning problem, which compiles away action contingency. 2) Solve the planning problem using an automated planner to obtain a solution plan. 3) Perform a dependency analysis on the plan to build a dependency graph (DP) and mark DP to generate a master plan P. 4) Project P to a distributed plan R based on the DP. We will also show that R is a distributed solution to the SCP, by showing that P is a centralized solution and that R is equivalent to P.

**Planning formulation of SCP**

A classical planning problem $\pi$ is a tuple $\pi = (A, F, I, G)$, where $A$ is the set of actions, $F$ the set of facts, $I$ the initial state, and $G$ the goal state. Each action $a$ has sets of preconditions $pre(a)$, add effects $add(a)$, and delete effects $del(a)$. The planning task is to find a sequence of applicable actions that transforms $I$ into a state containing $G$. 

\[ R ::= op \quad (op \in w) \mid send(ch,i,j) \quad (send to role j) \mid recv(ch,j,i) \quad (receive from role j) \mid R; R \quad (sequential) \mid R || R \quad (parallel) \mid c \triangleright P \text{ or } P \quad (c \in C, choice) \]

where $C \subseteq C$ 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 channel ID for each $send/recv$ pair. Like $talk$, $send$ and $recv$ are applicable to any service state.

**Definition 6 (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 $L \in C(R)$, $r_{in} \oplus L$ is defined and $r_{in} \oplus L \supseteq r_{out}$.

**Definition 7 (Equivalence).** A centralized solution is equivalent to a distributed solution when their sequence sets contain identical sequences, ignoring talk, send, and recv.

A closely related equivalence has been studied in (Qiu et al. 2007). We comment that WS-CDL can be viewed as an extended language for master plans, while MAP (Barker, Walton, and Robertson 2009) and the Role Language in (Qiu et al. 2007) are examples of languages for distributed plans.

### Automated Choreography by Planning

The initial choreography inputs $r_{in} = \{pid, pid\_name, pid\_quantity\}$. The goal is $r_{out} = \{purchase\_confirm, po\_order\_reject\}$, which has a purchase confirmation $purchase\_confirm$ and an order rejection $po\_order\_reject$. Note that here we assume that the user can specify multiple possible goal states he is interested in. Typically, the users are knowledgeable of these multiple choreography goals, because they are advanced model developers of Web service composition (Ponnkanti and Fox 2002). A choreography 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 choreographer can find plans contingent on the availability of the product.

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

\[
P ::= op \quad (op \in w \in W) \mid talk(i,j) \quad (role i talks to j) \mid P; P \quad (sequential) \mid P || P \quad (parallel) \mid c \triangleright P \text{ or } P \quad (c \in C, choice)
\]

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 (Barker, Walton, and Robertson 2009; Qiu et al. 2007), 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:

\[
t(op) = \{\{op\}\} \quad t(talk(i,j)) = \{\{talk(i,j)\}\} \quad t(P_1; P_2) = t(P_1) \circ t(P_2) \quad t(P_1 || P_2) = t(P_1) \triangleright t(P_2) \quad t(c \triangleright P_1 \text{ or } P_2) = \begin{cases} t(P_1), & \text{if } Q \triangleright c \triangleright P_1; \\ t(P_2), & \text{otherwise.} \end{cases}
\]

where $\circ$ denotes the concatenation of two trace sets (i.e. $A \circ B = \{(a,b) | a \in A, b \in B\}$), and $\triangleright$ is the interleaving of two trace sets (the definition is standard and omitted). $Q \triangleright c \triangleright op$ should be understood as $Q \triangleright c \triangleright op$ for any $op$ that can be the first operation in a sequence in $t(P_1)$.
Figure 2: The sequential plan for the running example (C: Customer; S: Supplier; W: Warehouse).

Given a SCP=(W,C,r_{in},r_{out}), we translate it into the following classical planning problem.

**Actions.** We develop actions in A in two steps. For each action \(a \in A\), we also define two properties: \(\text{key}(a)\) denotes the operation or contingency \(a\) is representing; and \(\text{host}(a)\) denotes the Web service \(w\) that \(\text{key}(a)\) belongs to.

1) For each Web service \(w \in W\), for each operation \(op \in w\), we introduce an **operation action** \(a\) with \(\text{pre}(a)=\text{op.I}\) and \(\text{add}(a)=\text{op.O}\). We also define \(\text{key}(a)=\text{op}\) and \(\text{host}(a)=w\).

2) For each contingency \(c \in C\), let \(c=(op,I',pre')\), we introduce a **contingency action** \(b_c\) and define \(\text{pre}(b_c)=[I]\), \(\text{add}(b_c)=[\text{cont}_c]\), \(\text{key}(b_c)=\text{op}\), and \(\text{host}(b_c)=w\) where \(op \in w\). Here, \(\text{cont}_c\) is a special fact introduced for each contingency \(c\). We also modify the action \(a\) for \(op\) by \(\text{pre}(a)=\text{pre}(a) \cup \{\text{op.I}\}\) \(\setminus \{\text{pre}_i\}\) \(\cup \{\text{cont}_c\}\). This way, we compile away the contingency on action outcomes.

**Facts.** The facts include all of the input and output parameters \((\text{op.I} \text{ and } \text{op.O})\) for every operation \(op\) in all services \(w \in W\), and all of the special contingency fact \(\text{cont}_c\), for \(c \in C\).

**Initial and goal states.** We set \(I=r_{in}\) and \(G=r_{out}\).

**Example 2.** The SCP shown in Example 1 is translated into a planning problem \(\pi\) with \(A = \{\text{ReceiveRFQ, ConfirmPO, \ldots, b}_1, b_2\}\).

For action \(a\) from operation \(\text{ReceiveRFQ}\), \(\text{pre}(a) = \{\text{pid, pid.name}\}, \text{add}(a) = \{\text{pid.price}\}\). The action \(b_1\) for contingency \(c_1=(\text{ConfirmPO, po.avail, \{avail\}})\) has \(\text{pre}(b_1) = \{\text{po.avail}\}\) and \(\text{add}(b_1) = \{\text{cont}_1\}\). The action \(a'\) for \(\text{ConfirmPO}\) has \(\text{pre}(a') = \{\text{cont}_1\}\) and \(\text{add}(a') = \{\text{po.accept, shiporder.R}\}\).

\(F = \{\text{pid, pid.name, po.avail, \cdots, cont}_1, \text{cont}_2\}\), \(I = \{\text{pid, pid.name, pid.quantity}\}\), and finally, \(G = \{\text{purchase, confirm, po.order_reject}\}\).

After compilation, we use a planner to find a solution plan for \(\pi\), which is a sequence of actions \(A=(a_1, \cdots, a_m)\) transforming \(I\) to \(G\). A solution to \(\pi\) in Example 2 found by FF (Hoffmann and Nebel 2001) is in Figure 2.

**Generation of the master plan**

Given a solution plan \(A=(a_1, \cdots, a_m)\), we build a dependency graph as follows.

**Definition 8 (Dependency).** Given a solution plan \(A=(a_1, \cdots, a_m)\), an action \(a_i\) **depends on** \(a_j\) (denoted as \(a_i \models a_j\)) if and only if, \(i < j\) and there exists a fact \(f \in \text{pre}(a_j)\) such that \(f \notin I\) and \(a_i\) is the last action in \(a_1, \cdots, a_{j-1}\) such that \(f \in \text{add}(a_i)\).

**Figure 3:** The DG for the example. The goal states \((G_1\text{ and } G_2)\) are also added to the graph for better illustration.

Note that the above definition is general for both operation actions and contingency actions.

**Definition 9 (Dependency Graph (DG)).** Given a solution plan \(A=(a_1, \cdots, a_m)\), a dependency graph is a directed graph \(DG=(V,E)\) such that \(V=A\) and there is an edge \((a_i,a_j)\in E\) if and only if \(a_i \models a_j\).

**Example 3.** Figure 3 shows the DG for the plan in Figure 2. The DG discovers structures, including sequential, parallel and conditional ones, of the master plan. We observe that there is a choice of making a decision based on contingency actions \((b_1, b_2)\) after the Supplier receives the availability information of an order. Moreover, once a product order is accepted \((\text{ConfirmPO})\), roles work in parallel in order to make payment and arrange shipping.

A choreography master plan can be derived from the DG.

- For every operation action \(a\) in the DG whose out degree is 1, we mark \(a\) by “\(\cdot\)”.

- For every contingency action \(b_c\) in the DG, we mark it by the contingency sign “\(\parallel\)”.

- For every action \(a\) whose out degree is more than 1, we mark \(a\) by “\(\|\)” if its successors do not include contingency action and mark \(a\) by “or” otherwise.

- For every edge \((a_i,a_j)\), if \(\text{host}(a_i) \neq \text{host}(a_j)\), we mark the edge by \(\text{talk}(i, j)\).

After such marking, we can write out the choreography master plan by viewing the marked DG as the parsing graph for the master plan language.

**Example 4.** Figure 3 shows the marking of DG (talk’s are not shown). The master plan derived from the DG is:

- \(\text{ReceiveRFQ}; \text{talk}(S,C); \text{ReceiveQuote}; \cdots; (c_1\parallel (\text{ConfirmPO}; \text{talk}(S,C); \text{ReceivePOAccept}; \cdots)) \parallel (\text{talk}(S,W); \text{ReceiveShipOrder}; \cdots); \text{ReceiveShipPayment}; \text{talk}(S,C); \text{ReceiveBuyConfirm})\) or \((c_2\parallel (\text{CancelPO}; \text{talk}(S,C); \text{ReceivePOReject})\)

**Generation of the distributed plan**

To generate the 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, \(V = V_1 \cup \cdots \cup V_k\), where \(a \in V_i\) if and only if \(\text{host}(a) = w_i\). It is illustrated in Figure 4.

Since \(a_i \models a_j\) only when \(i < j\), the DG is acyclic. Hence, a node \(a_j\) is an **offspring of** \(a_i\) if there is a path from \(a_i\) to \(a_j\) in DG. We define that an action \(a_i\) **depends on** a contingency \(c\) if \(a_i\) is an offspring of \(b_c\) in DG.
Definition 10 (Lead Operation) Given a solution sequence $A = (a_1, \ldots, a_m)$, for a given contingency $c \in C$ and role $w \in W$, the lead operation lead$(c,w)$ is the first action $a$ in $A$ 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 marked DG, for every role we generate a local plan in the language $R$ defined in Section 3. The entire algorithm is long and omitted, and we give the key 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 $A$.

- 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 ("\(\Vert\)") if they depend on the same contingency or by choice ("or") otherwise.

- For every edge from an operation action in $V_k$ to an action in a different partition $V_i$, we insert send$(ch_i,k,l)$.

- For every edge from an operation action in a different partition $V_i$ to an action in $V_k$, we insert recv$(ch_i,l,k)$.

- For every contingency action $b_k$ in $V_i$, we insert send$(ch_i,k,l)$ to every other partition $V_j$ that contains a lead operation.

- For every action $a$ such that $a = lead(c,w_k)$, where host$(b_k) = l, l \neq k$, we insert recv$(ch_i,k,l)$ before $a$.

Example 5. Based on the partitioned DG in Figure 4, the local plans are, for Customer: recv$(ch0,S,C)$; ReceiveQuote; send$(ch1,C,S)$; recv$(ch2,S,C)$; recv$(ch3,S,C)$; SendPOReject or (recv$(ch4,S,C)$); ReceivePOAccept; MakePayment; send$(ch5,S,C)$ || (recv$(ch6,W,C)$); ReceiveShipDetails; send$(ch7,C,W)$); recv$(ch8,S,C)$; ReceiveBuyConfirm.

Supplier: ReceiveRFQ; send$(ch0,S,C)$; recv$(ch1,C,S)$; ReceivePO; send$(ch10,S,W)$; recv$(ch11,W,S)$; ReceiveAvail; (send$(ch2,S,C)$ || send$(ch8,W,S)$); (recv$(ch3,S,C)$ || ConfirmPO; send$(ch4,S,C)$ || send$(ch12,S,W)$); (recv$(ch5,C,S)$ || recv$(ch9,W,S)$); ReceiveShipPayment; send$(ch8,S,C)$.

Warehouse: recv$(ch10,S,W)$; ReceiveQI; CheckAvail; send$(ch11,W,S)$; recv$(ch8,S,W)$; (recv$(ch12,S,W)$; (recv$(ch12,S,W)$); (recv$(ch12,S,W)$); (recv$(ch12,S,W)$); (recv$(ch9,W,S)$).

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

Theorem 1. For a SCP, the distributed plan generated by our algorithm is equivalent to the master plan.

Due to space limit, we only sketch the key idea of proof here. Our proof constructs an one-to-one correspondence between any possible sequence in the master plan to a sequence in the distributed plan, ignoring the communication primitives. (Qiu et al. 2007) 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 in our approach, only those master plans and local plans that can be generated from our planning method are considered. Our planning method, although more restrictive, ensures the equivalence of local and master plans by analyzing the dependency and contingency and using communication to enforce actions’ partial orders.

A counterexample in (Qiu et al. 2007) is a master plan $(a_1^1 \parallel a_2^1; a_2^1)$, where $a_1^1$ and $a_2^1$ are in role 1 and $a_2^2$ in role 2. The distributed plan is $(a_1^1; a_2^1)$ for role 1 and $a_2^2$ for role 2, which allows $(a_1^1; a_2^1; a_2^2)$, a sequence the master plan does not allow. In our planning approach, however, such a non-equivalence will not occur. If $a_1^1$ depends on $a_2^1$, then there will be a send/recv pair that restricts the order; if $a_2^1$ does not depend on $a_2^2$, then the master plan will be $(a_1^1; a_2^2) \parallel a_2^2$, which is equivalent to the distributed plan.

In addition, under contingencies, it is suggested that equivalence relies on the existence a dominant role (Qiu et al. 2007), 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.

Experimental Results

We developed a prototype system in Java. It takes service repositories in WSDL as input, allows user to specify initial state and choreography goals, and applies our approach to
generate a distributed choreography plan in the MAP specification (Barker, Walton, and Robertson 2009). Two planners FF (Hoffmann and Nebel 2001) and SatPlan06 (Kautz, Selman, and Hoffmann 2006) are integrated in our system. We ran our experiments on a PC with Intel Pentium(R) dual core processor 2.4 GHz and 1G RAM.

Currently a user needs to specify contingencies the first time 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 have enhanced WSDL by some special annotations to describe the contingencies. Since typically only a small number of Web services involve contingencies, it does not require much work as most Web services do not need any change. 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 annotations for contingencies and correctly translate them into contingency actions.

Taking the choreography dependency graph in Figure 3 as an example, we convert it into three MAP specifications. Figure 5 shows the MAP plan for Customer. Supplier and Warehouse are omitted. The translation from our language R to MAP is direct from the example. Details of MAP can be found in (Barker, Walton, and Robertson 2009).

We tested and compared our approach with WSPR (Oh, Lee, and Kumara 2007) on 81,464 Web services in 18 groups of large scale service repositories from the ICEBE05 Web services composition challenge. We compared against WSPR since it is a well-known solver for automated composition of Web services using AI planning techniques, i.e., forward search and regression search algorithm. Note that WSPR solves orchestration, a much simpler problem than the choreography problem that our approach solves. Since the most time-consuming components in our approach are SCP translation and choreography plan generation, we report the timing of these two parts in Table 1. The response time of finding a choreography plan is the duration from submitting a choreography task until receiving the output of the final solution or failing to find a plan.

Although the most difficult 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. Each time when a user wants to find a service choreography, he only needs to specify the inputs and possible 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, we do not need to parse a service repository again when a user submits a choreography task.

Comparing to WSPR, we can see that although our approach solves the more complex choreography problem and handles distributed coordination, communication, and contingency, it is more efficient than WSPR in finding a solution plan on 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.

### Conclusions

Automatic and efficient Web service composition (WSC) can potentially simplify the implementation of business processes. Web service choreography is an important paradigm for WSC with many advantages over Web service orchestration. This paper presents a method to generate distributed plans for Web service choreography. Based on an AI planning approach and dependency graph analysis, the method is efficient and fully automated, and also ensures correct collaboration of multiple peers.

In this paper, we focus on finding a feasible choreography plan that can correctly achieve choreography goals under distributed collaboration and contingencies. For our future work, we will further consider the optimization of choreography plan quality, by considering important non-functional property metrics, such as multiple QoS of Web services.

### Acknowledgment

We would like to thank Jörg Hoffmann, Henry Kautz and Bart Selman for providing open source of Metric-FF and SatPlan06. This work was supported by an NSF grant IIS-0713109, a Microsoft Research New Faculty Fellowship, a Shanghai Leading Academic Discipline Project grant J50103 and an Innovation Program of Shanghai Municipal Education Commission grant 11ZZ85.
References