A QSQL-based Efficient Planning Algorithm for Fully-automated Service Composition in Dynamic Service Environments

Kaijun Ren\textsuperscript{1,2}, Xiao Liu\textsuperscript{2}, Jinjun Chen\textsuperscript{2}, Nong Xiao\textsuperscript{1}, Junqiang Song\textsuperscript{1}, Weimin Zhang\textsuperscript{1}

\textsuperscript{1}College of Computer, National University of Defense Technology, Changsha, Hunan 410073, P.R. China
renkaijun@nudt.edu.cn

\textsuperscript{2}Centre for Information Technology Research, Swinburne University of Technology, Melbourne 3122, Australia
jchen@swin.edu.au

Abstract

Web service composition is emerging as a promising technology for supporting large-scale, sophisticated business process integration in a variety of complex e-science or e-business domains. Particularly, semantics have been proposed as a key to automatically solving the discovery and composition problem. However, most of semantic composition approaches still remain at a stage of low efficiency because of the performance issues brought by the involved ontology reasoning and manual processing.

To address this problem, in this paper, we present a QSQL-based service composition algorithm towards a fully-automated service composition. QSQL (Quick Service Query List) is an efficient service query index list which can achieve about the same semantic service discovery effects as other existing semantic composition methods, but with much less reasoning. With our proposed QSQL-based service composition algorithm, composition plans can be created to meet a user’s query in an automatic, efficient and semantic manner. In particular, with our algorithm, most existing composition plans in QSQL can be founded and ranked by exploiting a weighted Petri net representation; which will facilitate the execution verification. The final experiment is conducted to further demonstrate the feasibility of our proposed composition approach and its efficiency.

1. Introduction

Web service composition, by binding two or more existing services into a new one, is emerging as a promising technology for supporting large-scale, sophisticated business process integration in a variety of complex e-science or e-business domains. With the increase in importance of business process automation and highly dynamic nature of the internet, automated composition approaches have been the most promising methods to be applied in real world contexts. Currently, there exists many different research efforts on automating service composition; such as Petri net-based composition approaches [1, 2], AI planning-based approaches [3, 4], context and non-functional properties-based approaches [5-7]. But most importantly, semantic service compositions that take semantics of services into account to automatically solve the discovery and composition problem, have been a recent active research field [8-10]. However, despite the merits and the importance of semantic information contained by services, most existing algorithms for semantic service composition have taken direct ontology reasoning for generating process plans, which have caused performance issues. For example, [11] provided a hybrid semantic service discovery method based on direct ontology reasoning on top of OWL-S described services. The provided examples contain 582 services, 29 query requests and the average response time for each query is about 8 seconds when being simulated in the computer with a 2.4 GHz CPU, 1024MB memory. Therefore, if there is no single web service satisfying the requests, a process plan by composing multiple services has to be generated, thus the response time will be much longer. As such, the traditional semantic composition approaches involving much reasoning are unable to offer the capabilities of quick response for user’ query when he demands a quick response in highly dynamic internet service environments. Additionally, with the growing service resources, manual processes in some service composition approaches can be another
prohibition toward fully-automated service composition.

To overcome the aforementioned problems, in this paper, we present a QSQL-based composition approach for quickly and automatically creating process models for resolving dynamic distributed application integration problems. QSQL (Quick Service Query List) is an efficient service query index list where the important reasoning relationships and service information can be processed and recorded in the specially designed data structures during service publication in advance. Further, based on QSQL, our proposed composition algorithm can not only ensure a quick query response but also ensure a high semantic composition quality without involving much ontology reasoning and manual processing. In particular, with our methods, most composition plans existing in QSQL can be founded and ranked by employing the formal and weighted Petri net structures to meet user’

The remainder of this paper is structured as follows: Section 2 gives an analysis of related work. Section 3 gives a summary of QSQL principles. Section 4 provides a formal and weighted Petri net description for representing process constructs. Section 5 presents our concrete composition algorithm and its complex analysis. Section 6 presents the experiment comparison. The final section gives a conclusion and our future work.

2. Related Work and Problem Analysis

Over the last several years, different representative composition strategies have been proposed in the literatures. To the best of our knowledge, we categorize and analyze them below.

- Composition based on Petri nets:

  [1] Proposes a goal-driven service composition approach based on a Petri nets modeling technique. With this method, goals are recursively decomposed to be sub-goals until they become executable elementary goals. Because Petri net is a suitable tool for formal analysis and verification of correctness, this approach is of some significance. However, the author didn’t give the efficiency comparison of goal decomposition. In actuality, goal decomposition may be time-consuming tasks because of similar complex AI searching.

- Composition based on AI planning:

  Many research efforts regard the service composition problems as AI planning problems, where a planner is used to determine the combination of actions [3, 4]. With this approach, an explicit goal definition needs to be provided. However, such explicit goal is usually not available. Therefore, most of these approaches are restricted to sequential compositions. Additionally, performance issue can also present another problem when applying sophisticated AI algorithms for automated plan generation.

- Composition based on formal models: (Process algebra, Finite-State Machines)

  Some composition approaches are based on formal models such as process algebra [12], Finite-State Machine [13]. The above formal composition methods are very useful to check the existence of a composition and return a composition plan if one exists. However, they fail to automatically find most of such existing composition plans and rank them. In dynamic network environments, alternate execution plans are very important when exceptions occur.

- Composition based context and non-functional properties:

  Some composition approaches depend on context and non-functional properties of services. [5] proposes a dynamic and graph theory-based service composition approach aiming at capturing the useful characteristics of resources in a pervasive computing environment. However, in this method, the basic matching is still on the direct semantic reasoning level. Therefore, the performance problems for producing composition plans should also be overcome. Actually, most of these approaches did not pay much attention to the efficiency of automatically generating process flow.

- Composition based on semantic approaches

  Semantics have been proposed as a key to increasing automation in applying Web services and managing Web processes within and across enterprises, and the World Wide Web Consortium (W3C) has recently finished an important standard--Semantic Annotations for WSDL and XML Schema (SAWSDL) for Semantic Web services(SWSs)[14]. Currently, there are many existing semantic service composition approaches[8-10] and semantic composition projects such as METEOR-S[15], IRS[16] SHOP2[3], IBM semantic tools[17]. However, these methods still either stay in a semi-automated state, or in low efficiency of producing a composition plan due to a direct reasoning style.

  By comparison, our proposed QSQL-based service composition approach can overcome the performance issue by avoiding large scale of ontology reasoning so that a given query can be achieved a quick response during the creation of composition plans. Particularly, most of existing plans can also be founded and ranked by exploiting a formal and weighted Petri net representation for facilitating the execution verification.
3. Summary of Quick Service Query List (QSQL)

In order to overcome the low discovery efficiency brought by the traditional semantic service discovery algorithm based on direct reasoning, we have proposed a pre-reasoning based service discovery method in [18]. In this method, ontology reasoning can be done in advance when services are published, and the reasoning results can be recorded in specially designed data structures which are based on graph storage theory to form quick service query index called QSQL. The basic data structures of QSQL mainly include two parts. One part is the domain of link which is mainly used to avoid repeated reasoning. The other part is the domain of data, which is primarily used to record service information in the corresponding INPUT/OUTPUT data vectors such as Exact_vector, Plugin_vector, Sib_vector, Grapar_vector, Grachd_vector according to their corresponding semantic relationships. In order to clearly understand QSQL, let’s assume there are some web service models which need to be published to form QSQL in Table 1. These examples will also be used to exemplify the later composition algorithm. Particularly, the parameters of service models have been simplified to directly match the corresponding ontology concepts. Some semantic relationships between ontology concepts with respect to the relevant service models in Table 1 are as follows.

- USPrice ⊇ USdollar
- datetime ⊇ somedatetime
- somedatetime ⊇ givendatetime
- areacode ≡ cityid
- location ≡ cityname

Table 1. Abstract web service models

<table>
<thead>
<tr>
<th>ID</th>
<th>Operation</th>
<th>Input1</th>
<th>Input2</th>
<th>Input3</th>
<th>Output1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ws1</td>
<td>UStoRMB</td>
<td>USprice</td>
<td>givenDateTime</td>
<td>RMBPrice</td>
<td></td>
</tr>
<tr>
<td>ws2</td>
<td>GetStockPrice</td>
<td>cityID</td>
<td>stockID</td>
<td>USdollar</td>
<td></td>
</tr>
<tr>
<td>ws3</td>
<td>cityCode</td>
<td>cityName</td>
<td>stockID</td>
<td>citycode</td>
<td></td>
</tr>
<tr>
<td>ws4</td>
<td>getStocked</td>
<td>stockName</td>
<td></td>
<td>stocked</td>
<td></td>
</tr>
<tr>
<td>ws5</td>
<td>getregioncode</td>
<td>location</td>
<td>areacode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

According to the introduced service publication algorithm in [18], part publication records in QSQL for services in Table 1 are given in Table 2.

Table 2. Part records of QSQL for models in Table 1

<table>
<thead>
<tr>
<th>ID</th>
<th>Ontology</th>
<th>Domain</th>
<th>Input1</th>
<th>Input2</th>
<th>Input3</th>
<th>Output1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>price</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>USprice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>givenDateTime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>USdollar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>somedatetime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>cityid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>cityName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>stockID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>citycode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>stockName</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>datetime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>areacode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, when publishing

\[ ws5(getregioncode(input(location),output(areacode))) \]

areacode is an output parameter of ws5, the publication algorithm first inserted ontology concept areacode into QSQL, then added unique ID of ws5 to its Output-Exact_vector domain(ID 13 of Table 2). o.exact means Output-Exact_vector, similar for others; further the algorithm depended on ontology reasoning to find out all other related ontology classes such as equivalent, super, sub classes etc for areacode, then respectively added ID of ws5 to their corresponding output vectors. For instance, in the above semantic relationships, because of \[ \text{areacode} \supseteq \text{cityid} \], \[ \text{cityid} = \text{citycode} \], the algorithm inserted ontology concepts cityid and citycode into QSQL; then added ID of ws5 to their Output-Plugin_vector domains(ID 6, 9 of Table 2). Repeatedly, the publication algorithm processes each parameters of ws5. As such, all ontology concepts which have semantic relationships with WS5 could be found to record the information of ws5 in their corresponding vectors by algorithm. Therefore, QSQL can make sure a large number of ontology reasoning can be processed at service publication stage so that semantic service discovery and composition can be achieved a quick response. More details can be found in [18].

4. Extended Petri net Representation for Process Model

A Web service behavior is basically a partially ordered set of operations, and it is straightforward to map it into a Petri net in [1, 2]. In our methods, we extend the definition of Petri net and composition operators in [1, 2] so that our composition plans can be directly mapped by the weighted Petri net for dynamic creation of process flow.

**Definition 4.1 (Extended Basic Petri net (EBPN))**

An extended basic Petri net is a weighted service net SN with a 7 tuple \( \langle P, T, R, I, O, W, S \rangle \) where:

1. \( P \) is a finite set of places;
2. \( T \) is a finite set of transitions representing the operations of the service;
3. \( R \) is a finite set of arcs where \( R \subseteq \{P \times T \} \cup \{T \times P \}; \)
4. \( I \) is the input place; \( O \) is the output place;
5. \( W : R \rightarrow \{1, 2, 3, 4, 5\} \) is a weight function which will represent a corresponding matching degree according to the definition in [18]. Namely, 1 denotes Grachd; 2 denotes Grapar; 3 denotes Sib; 4 denotes Plugin; 5 denotes Exact;
6. \( S : T \rightarrow A \cup \{\Theta\} \) is a labeling function where \( A \) is a set of operations of all WS(1, 0, ...), \( \Theta \) denotes a null operation.

A composition process flow is composed of weighted basic service nets by effective composition operators. These composition operators are used to perform algebraic operations over services to obtain
composite services[1]. In this paper, we only give three basic composition operators: sequence, choice, and parallelism operator.

**Definition 4.2 Sequence Operators**

\( \triangleright \) denotes an operator of sequence. \( w_{s1} \triangleright \triangleright w_{s2} \) represents two adjacent services in a sequential relationship. Figure 1 shows the graphical representation of \( w_{s1} \triangleright \triangleright w_{s2} \).

![Figure 1. Sequence services \( w_{s1} \triangleright \triangleright w_{s2} \)](image1)

The corresponding net SN of \( w_{s1} \triangleright \triangleright w_{s2} \) is \( \{P,T,R,I,O,W,S\} \) where

- \( P = P_1 \cup P_2 \); \( T = T_1 \cup T_2 \);
- \( R = R_1 \cup R_2 \); \( I = i_1; O = o_1; W = W_1; S = S_1 \).

**Definition 4.3 Choice Operators**

\( \otimes \) is a choice operator which allows building a composite service out of two services. The component services linked by the choice operator are executed alternatively. Figure 1 shows the graphical representation of \( w_{s1} \otimes w_{s2} \).

![Figure 2. Choice services \( w_{s1} \otimes w_{s2} \)](image2)

The corresponding SN of \( w_{s1} \otimes w_{s2} \) is \( \{P,T,R,I,O,W,S\} \) where

- \( P = P_1 \cup P_2 \); \( T = T_o \cup T_i \);
- \( R = R_1 \cup R_2 \); \( I = i_o \); \( O = o_i \); \( W = W_o \cup W_i \); \( S = S_o \cup S_i \).

**Definition 4.4 Parallelism Operators**

\( \oplus \) is a parallelism operator. The component services linked by the parallelism operator are executed concurrently. Figure 3 shows the graphical representation of \( w_{s1} \oplus w_{s2} \).

![Figure 3. Parallelism services \( w_{s1} \oplus w_{s2} \)](image3)

The corresponding SN of \( w_{s1} \oplus w_{s2} \) is \( \{P,T,R,I,O,W,S\} \) where

- \( P = P_1 \cup P_2 \); \( T = T_o \cup T_i \);
- \( R = R_1 \cup R_2 \); \( I = i_o \); \( O = o_i \); \( W = W_o \cup W_i \); \( S = S_o \cup S_i \).

5. **QSQL-based Planning Algorithm**

5.1. **Algorithm Design**

In the previous paper [18], we have proposed service publication rules and algorithm to build up QSQL. Simultaneously, we also have presented a service discovery algorithm for searching a single service from QSQL to meet the user’s requirements. However, in real application, service composition is very useful when there is no single service to satisfy user’s request. In this section, we give an efficient composition algorithm to automatically combine multiple services from QSQL to meet user’s requirements.

We take service models in Table 1 as a composition example. For a given query \( wsr((\text{cityname},\text{stockname},\text{datetime}),\text{o}(\text{RMBprice})) \), there will be no solution for single service meeting user’s requirements. But intuitively, as shown in Figure 4, \( w_{s5}, w_{s4}, w_{s2} \) and \( w_{s1} \) can also be composed in some orders to meet the query. Alternately, \( w_{s5}, w_{s4}, w_{s2} \) and \( w_{s1} \) can also be another plan to meet the query with a different matching effect. In order to automatically offer such composition plans meeting user’s query, we designed and implemented a QSQL-based composition algorithm. In our composition method, the only thing users need to do is to specify a query including the inputs and outputs. Particularly, these inputs and outputs will be mapped to the corresponding ontology concepts. Actually, many semantic tools and methods have been proposed to help annotate semantic information to services[17, 19]. Therefore, for a given user’s query, we can assume that the specified inputs and outputs in queries have been mapped by ontology concepts.

![Figure 4. Intuitive composition plans for query](image4)
should be a part of a successful searching path. Therefore, the algorithm inserts the path (I', (i), subgoal, matchingweight) to the composition plan (step 6). Then the algorithm begins to pop and match the next sub goal (step 7) from the stack. However, if IC doesn’t semantically contain subgoal, it means this subgoal can’t be met by the user’s inputs. Hence the algorithm begins to search QSQL and retrieve the related service models to meet subgoal (step 9). For this step, because all published service models have been semantically recorded in output vectors V_s, x=1,2,3,4,5 (defined in Table 3) of each ontology concept in QSQL, the algorithm can easily retrieve all service models that match subgoal by finding its corresponding ontology concept from QSQL. If ∀ x, subgoal. V_s x is null, it means this subgoal is unreachable; further all related paths with this subgoal should be deleted from the composition plans (step 11). Contrarily, for any V_s x, if they are not null, all service models WS, which belong to subgoal. V_s x (x=1,2,3,4,5) can sequentially match the sub goal, so the algorithm adds the path (WS, subgoal, x) to the composition path with the weight value x (step 14). Notably, x simultaneously represents the meaning of the matching degree according to the definition of Table 3. For example, if WS, (I', O'), UID ∈ subgoal. V_s x, then (WS, subgoal, 5) shows WS meets subgoal with the matching degree EXACT. Now back to the algorithm, if ∀ i, j, x, WS, (I', O'), UID ∈ subgoal. V_s x , WS, (I', O'), UID ∈ subgoal. V_s x , then the relationship between WS, and WS, is WS ⊕ WS, which means they can be the alternate path to meet this subgoal (step 15). In particular, for each WS, (I', O'), UID ∈ subgoal. V_s x , the inputs I' needs to be concurrently met by either user inputs, or the outputs of other service models so that WS, can be successfully executed to meet subgoal .

Therefore, the inputs I' of WS, should be pushed into the stack to be next searching goals (step 16). Considering the concurrent, these new sub goals should be added to the path of composition plans in parallelism order (step 17). Similarly, all sub goals should be popped from the stack and processed by step 3 to 17 until the stack is null. When the stack is null, either all composition plans in QSQL will be generated, or there is no such plans to meet the user’s query. Because the algorithm has taken the extended and weighted Petri net model, all final composition plans can be represented by Petri nets.

### Algorithm 1: FindAllProcessPlans

- **Inputs**: WS, subgoal, QSQL.
- **Outputs**: composition plans

1. For I', IC ← extendingInputSource(I');
   - Forming extended input source I' = I'; by user renaming
2. push( O');
   - (For each output o ∈ O', pushing o to stack so that it will be a search goal)
3. subgoal ← pop();
   - Retrieving a subgoal from target stack;
4. relationJudging( IC, subgoal );
   - Judging the semantic relations if IC ⊇ subgoal;
5. Case 1(true): IC ⊇ subgoal
6. addpath( I', (i), subgoal, matchingweight );
   - (If subgoal has been founded, adding path from concept of I' to subgoal, inserting the matching weight value on the path)
7. GO TO 3 ; (Popping next subgoal);
8. Case 1(false): IC ⊈ subgoal
9. V_s ← retrieveAllO_vectorsfromQSQL(subgoal), x = 1,2,3,4,5;
   - (Retrieving all output vectors V_s of subgoal from QSQL)
10. If ∀ x, subgoal. V_s x = ⊕ Then
11. deleteLocalPath(subgoal);
   - (It is impossible for subgoal to be arrived. Deleting the paths which are related with subgoal)
12. GO TO 3 ;
13. Else For V_s x, WS,(I', O'), UID ∈ V_s, x=1,2,3,4,5 Do
14. insertSequenceOperatorPath(WS, subgoal, x);
15. insertChoiceOperatorPath(WS, subgoal, O');
   - (Because of meeting the same subgoal, existing relationship WS, O', and pushing I' );
16. For i, j, x, WS,(I', O'), UID ∈ V_s, x=1,2,3,4,5 Do
17. insertParallelismOperatorPath(WS, subgoal, x);
   - (For inputs I' of WS, they need to be matched simultaneously to success WS, x)
18. EndFor
EndCase 2
EndElse
EndCase 1
EndGO

According to algorithm 1, our proposed composition algorithm mainly achieved the following distinguished advantages:

- For a given query, the algorithm can automatically find all composition plans from QSQL;
- All composition plans can be ranked by their weighted matching degree;
- All generated plans can be built efficiently without user interaction and much ontology reasoning;
- All generated plans can be represented by extended Petri nets which are formally introduced in section 4.

### 5.2. Complexity of Algorithm

As shown in algorithm 1, the computing complexity mainly depends on the scale of searching goals in stack and the cycle numbers between step 13 and 18. The number of searching goals will be changed
dynamically. Let’s consider the worst case, assume the total number of published service models is $N$, the maximum number of input parameters for each service model is $\mu$, then the upper limit of goals in stack is $\mu N + |\alpha'|$. ( $|\alpha'|$ : requested output number for query ). Additionally the maximum scanning number between step 13 and 18 is $5N$. Consequently, the total cycle numbers is $(\mu N + |\alpha'|) + 5N$. Normally, $\mu$ and $|\alpha'|$ are constants. Therefore, the total complexity of our composition algorithm is $O(N^2)$ under polynomial time. In particular, with our algorithm, the reasoning time happens only at the beginning when forming the extended input set $IC'$. Nevertheless, in the cycle body, there is not any reasoning happening because QSQL has stored semantic relationships between concepts and services in advance. By comparison, semantic reasoning is normally a key factor for performance issue to generate composition plans in the traditional semantic service composition approaches. We will further illustrate this in our experiment analysis.

5.3. Composition Example

For a given composition example in Table 1 and QSQL records in Table 2, we use our composition algorithm to find all composition plans to meet the specified query $wsr(i(cityname,stockname,datetime),o(RMBprice))$. The algorithm first formed the extended input source $IC'$ for query’s inputs(cityname,stockname,datetime) by racer reasoning such as $datetime \supseteq \text{somedatetime}$ (step 1), then pushed the query’s output $RMBprice$ into the stack to be searching goal(step 2). The algorithm began to pop a search goal $RMBprice$ from the stack (step 3), then judged if $IC' \supseteq RMBprice$ (step 4). Because $IC'$ didn’t contain $RMBprice$ , step 9 was executed. The algorithm found the matched ontology concept $rmbprice$ in QSQL, then retrieved the related service model from the output vectors of $rmbprice$. As we know from QSQL in Table 2, $rmbprice,V'_i = \{ws_j\}$ (ID 1 of Table 2 ), so $ws_j$ matched the goal $rmbprice$ with matching degree 5(EXIT). The algorithm then added the path ($ws_j,G_i : RMBprice,5$) (step 14) to the composition plans (as shown in Figure 5). Further, for the execution of $ws_j$ , the inputs $\{\text{givenDatetime}, \text{USprice}\}$ of $ws_j$ should be met concurrently by either query’s inputs or other service model’s outputs. Therefore, the algorithm pushed inputs $\text{givenDatetime}$ and $\text{USprice}$ into the stack to be next sub searching goals (step 16). Simultaneously, according to step 17, the path $(G_{i_1} : \text{USprice},ws_j,5)$ and $(G_{i_2} : \text{givenDatetime},ws_j,5)$ should be added to the composition plan, and the operation relationship between $G_{i_1}$ and $G_{i_2}$ is $G_{i_1} \otimes G_{i_2}$. Then the algorithm went back to step 3 to pop the next sub goal $USprice$. Similarly, because $USprice \subset IC'$, $USprice,V'_i = \{ws_j\}$ (ID 2 of Table 2), ($ws,G_i : USprice,5$) should be added to the composition plan, and the inputs (cityID, stockID, someDatetime ) of $ws_j$ were pushed into the stack to be next searching goals. Repeatedly, the searching goals in the stack were processed by the algorithm. Finally, when the stack is null, all weighted composition plans are generated and ranked. With this example, Figure 5 gives the whole processes for a given query $wsr(i(cityname,stockname,datetime),o(RMBprice))$. As shown in Figure 5, the final generated composition plans are ($ws_j \oplus (ws_j \oplus ws_j)$) $\rightarrow ws_j \oplus ws_j$ , and ($ws_j \oplus ws_j$) $\rightarrow ws_j \oplus ws_j$ to meet the query. At the same time, according to the former introduced weight computing method of section 4, the total weight value of ($ws_j \oplus ws_j$) $\rightarrow ws_j \oplus ws_j$ is $36(5+5+4+5+5+5+2+5)$, while ($ws_j \oplus ws_j$) $\rightarrow ws_j \oplus ws_j$ is $35(5+5+4+5+4+5+2+5)$.
Therefore, \((ws_1 \oplus ws_2) \rightarrow ws_3 \rightarrow ws_4\) is better than 
\((ws_2 \oplus ws_3) \rightarrow ws_4 \rightarrow ws_5\). However, if \((ws_1 \oplus ws_2) \rightarrow ws_3 \rightarrow ws_4\) fails to meet the user’s requirements, the plan 
\((ws_2 \oplus ws_3) \rightarrow ws_4 \rightarrow ws_5\) will become an alternate one.

6. Experiment and Evaluation

The simulation was performed in our real-world grid workflow management system called SwinDeW-G [21] to demonstrate the efficiency of the above algorithm. Particularly, we mainly focus on the comparison and analysis of the performance for producing composition plans without including the performance comparison of running actual composition plans by taking separating techniques between abstract composition plans and concrete composition execution plans. Currently, we have produced and published 3000 service models to QSQL by using six selected ontology domains [22]. In addition, we also produced 30 composition queries. Query client, QSQL database, and service repository are distributed in different nodes. In our experiment, we mainly compared the performance including composition time, memory consuming between QSQL-based composition methods and the traditional direct reasoning-based composition methods.

Figure 6. The comparison for composition query time
Composition time: Figure 6 shows the execution time for each query by using two different composition methods. As shown in Figure 6, the following distinctions can be easily achieved. First, the average execution time (11544ms) of each query by direct reasoning-based composition methods is higher than the time by QSQL-based composition method (2297ms). Secondly, the fluctuation of each query by direct reasoning is obvious. Thirdly, the execution time by direct reasoning-based composition method appears to an increasing trend. For the first distinction, the reason is that the direct reasoning-based composition methods need to involve much ontology reasoning when creating the process flow to meet the user’s query. By comparison, because the large number of ontology reasoning has been processed previously in QSQL, our algorithm can offer a quick response for each query. For the second distinction, different queries contained different numbers of ontology reasoning by direct reasoning-based methods, which directly caused to the fluctuation of composition time. While in our composition algorithm, ontology reasoning only happened in the beginning when forming the extended user’s input collection, thus the composition time is relatively stable. For the third distinction, next section can give a good explanation.

Memory Consumption: In our experiments, the memory consumption has been monitored and analyzed. Figure 7 and Figure 8 give the comparison results. Apparently, when the traditional direct reasoning composition algorithm continuously processed queries, more ontology will be loaded and classified which means more memory space is needed. When RAM is limited, the garbage collection engine in JVM had to work to release more memory which caused higher latency (Figure 8). This also proved the above third distinction. Therefore, from both sides of memory consumption and response time, the direct reasoning-based composition method has become unbearable. By comparison, our QSQL-based composition method can continually process queries in more stable way without the need of much reasoning.
Because the publishing time is not the critical time for application, QSQL-based composition methods are more efficient and meaningful.

7. Conclusions and Future

In this paper, we have presented a new efficient QSQL-based composition algorithm for automatic service composition. For a given user query, the proposed algorithm can not only ensure all existing composition plans in QSQL can be efficiently found in an automatic and semantic manner, but also can rank them by exploiting a weighted Petri net representation for further verification. Our experiments and valuations have demonstrated that our proposed QSQL-based composition algorithm is more efficient than other typical ones. In future, we will focus on how to dynamically bind service instances to abstract composition plans for making resource’s sharing more flexible.

8. Acknowledgement

We are grateful for the English proofreading by D. May, and the foundation support by the National “973” Research Fund Plan Foundation of China under Grant No. 2003CB317008, by National Nature Science Foundation of China under Grant No. 60573135, 40505023 and 60736013, and by Swinburne Dean’s Collaborative Grants Scheme 2007-2008, and by Swinburne Research Development Scheme 2008.

9. References