# Inheritance-Aware Document-Driven Service Composition

Zhifeng Gu, Bin Xu, Juanzi Li
Department of Computer Science and Technology
Tsinghua University Beijing, 100084, China
{gzf, xubin, ljz}@keg.cs.tsinghua.edu.cn

## Abstract

*In this paper, we proposed a fast service composition method to solve the semantic composition problem defined by WS-Challenge. We construct an inverted table to index the services according to their output attributes. Based on the index, we developed the algorithms for service discovery and composition. Thanks to the high speed of service lookup through the index, our algorithm gains a high performance.*

## 1 Introduction

Service Oriented Computing (SOC), which has been widely accepted as the next generation programming paradigm, defines promising technologies that enable future computing models over the Internet. In the SOC paradigm, services, being self-describing, open components that support rapid, low-cost composition of distributed applications, are fundamental elements to the development of applications [5]. Web services are a typical SOC example [3], and have been widely accepted and deployed in real world applications. In this paper, the term "service" and "web service" will be used interchangeably. There are many research topics about services, including service description, service discovery, service selection, service composition, service execution and monitoring, etc. Among these topics, service composition plays an central role.

It has become an acknowledgment in the research community that service composition could be classified into two categories: service orchestration and service choreography, which are shown in figure 1 respectively. The goal of service orchestration is to construct new services or applications with existing services. It has a central controller that invokes services exposed by different peers. The peers in an orchestration are independent with each other. Furthermore, service orchestration is usually executable. Unlike service orchestration, service choreography is usually unexecutable. It is the description of a system which consists of several peers. The peers in a choreography are designed to communicate with each other, and to make the system run correctly. They can be implemented as service orchestrations or some other technologies such as J2EE and .NET. As most of the works about service orchestration are still using the term "service composition", we will also use "service composition" to indicate service orchestration in the following text.

In this paper, by service discovery, we mean a process to find services that can produce a required document; by service composition, we mean a process to construct a service chain that finally produces the required documents (the outputs) and can be satisfied with the available documents (the inputs). We propose a quick algorithm to search services and construct a service composition according to the given input/output documents. Our focus is especially on the efficiency of the algorithm.

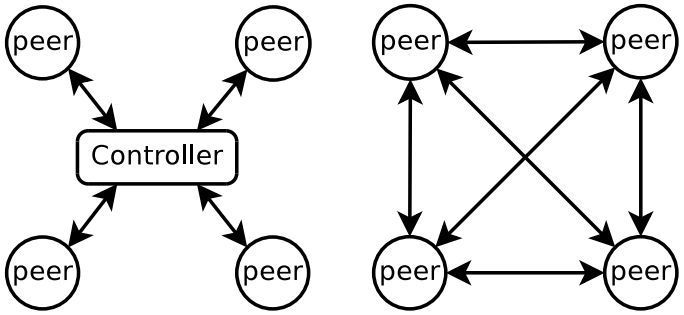
## 2 Related Work

Input and output documents is an important feature of services. Many works have been proposed based on the I/O of services.

SWORD [6] models web services with conditional input/output and data input/output. It uses a rule-based expert system to automatically generate plan for service composition according to the I/O of services. Liang [2] proposed an semi-automated method to construct service composition. The key idea of her work is the service dependency graph (SDG). SDG is a directed graph that shows all the possible input/output dependencies among different services. Liang has given a algorithm to construct service composition templates by searching through an AND/OR graph derived from SDG. This method is similar to the syntactic composition defined by WS-Challenge [1], which is a purely document-driven method for service composition. The difference is that Liang's method allows user to specify the services that must be invoked.

[1] http://www.ws-challenge.org/

The 9th IEEE International Conference on E-Commerce
Technology and The 4th IEEE International Conference
on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 © 2007 IEEE




(a) orchestration (b) choreography

**Figure 1. Service composition**

As well as the syntactic composition, WS-Challenge also defines the semantic composition which is an enhanced, inheritance-aware version of syntactic composition. Various methods aiming at solving the syntactic composition and semantic composition problem at a high speed have been proposed, e.g. [1] [4]. Our work is one of the competitors that try to solve the semantic composition problem. Compared with our previous work in [7], the new version has the following enhancements:

- fully support semantic composition, while the previous version may crash when the data set is large.
- re-implemented with C++, while the previous version is implemented with C#.
- many code level tuning have been done to improve the performance.

It is expectable that the new version will get a higher score than the old version.

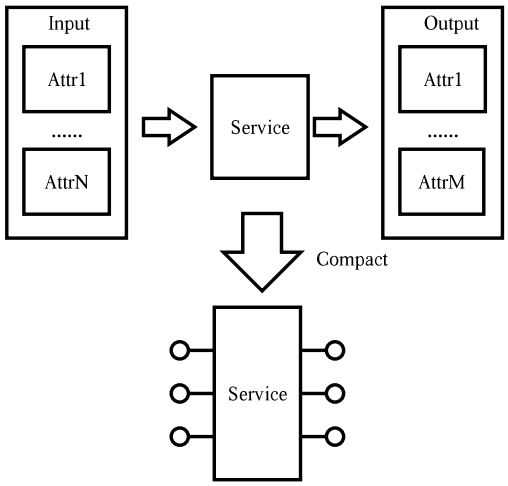



**Figure 2. The service model**

## 3 Our Approach

### 3.1 Concepts and Definitions

Before introducing the composition algorithm, we first formally define the concepts that will be referred to in the following sections.

Figure 2 gives an illustration of the service model that is used in this paper. In the service model, a service is considered as a black box. It consumes a message and produces another message. A message consists of several attributes, and each attribute has a data type associated with it. For convenience, we will use a compact symbol, the one at the bottom of figure 2, to represent services. Compared with the underlying service model of WSDL, this model ignores the concept of operation. However, as the interface defined WSDL is stateless, this reduction does not reduce the expressiveness.

Thus, a service $WS$ is formally defined as,

**Definition 1** $WS = \{D_{in}, D_{out}\}$, where
$D_{in} = \{d_i \mid d_i$ is an attribute of the input data$\}$
$D_{out} = \{d_i \mid d_i$ is an attribute of the output data$\}$

Accordingly, a request $RQ$ is defined as,

**Definition 2** $RQ = \{R_{in}, R_{out}\}$, where
$R_{in} = \{d_i \mid d_i$ is an input attribute provided by the user$\}$
$R_{out} = \{d_i \mid d_i$ is an output attribute required by the user$\}$

The task to find services satisfying a given request $RQ$ can be formally described as to find the services that hold the following formulas.

$D_{in} \subseteq R_{in}$ and $D_{out} \supseteq R_{out}$

### 3.2 Indexing

In our method, indexing is the key point to accelerate the composition algorithm. The index of services is con-



The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 © 2007 IEEE

structed as an inverted table. As illustrated in figure 3, the inverted table consists of two parts: the output "vocabulary" and "occurrences". The output "vocabulary" is a set of all the output attributes of the services in a given set. Each attribute is a key, and has an list of services associated with it. The services in the list must hold the following conditions:

- Either the output set of the service contains the key attribute: $d_{key} \in D_{out}$,
- or the output set of the service contains one of the attributes that are directly or indirectly inherited from the attribute associated with the list.

Simply speaking, the list contains all the services that can produce the attribute associated with the list. Such a list is so-called the "occurrences" in the inverted-table. In our implementation, each attributes is mapped to an integer ID to improve the performance.

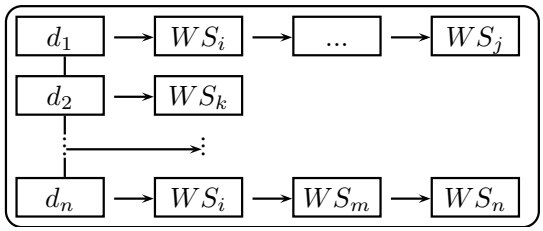



**Figure 3. The inverted table for indexing**

### 3.3 Algorithms

The composition task is to find a service chain that can fulfill the user request. In the chain illustrated in figure 4, the input attributes of each service should be fulfilled by the output attributes of the precedent service or the input attributes of the request. Obviously, the input attributes of the first service of the chain must be in the input attributes of the request; and output attributes of last service must contain the output attributes of the request.

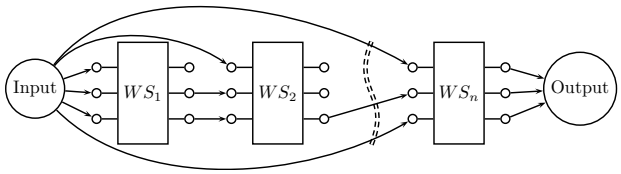



**Figure 4. A service chain**

A service chain is constructed from the output set of the request. Firstly we find the last service $WS_n$, whose output set contains the output set of the request, i.e. $D_{out}(WS_n) \supseteq R_{out}$. Secondly we will find service $WS_{n-1}$ that can produce the attributes belonging to the set difference $D_{in}(WS_n) \setminus R_{in}$. Then, do the second step repeatedly until the input set of the service can be covered by the input set of the request, i.e. $D_{in}(WS_i) \subseteq R_{in}$. Since we have encoded the inheritance relationships in the index, the composition algorithm does not need to deal with the inheritance issue again, which makes it nearly the same as the algorithm for syntactic composition. The algorithm for service discovery and service composition is formally given in algorithm 1 and 2 respectively. In the service composition algorithm, "getServices" is a method that finds all the services whose outputs can cover the set difference of the given output set and the input set of the request. As the internal logic of "getServices" is quite similar to the service discovery algorithm, we do not describe it in detail. "TraceChain" is a function that will trace all the service chains recursively. It will generate the output in the required XML format.

**Algorithm 1** Algorithm for Service Discovery

**Input:** The request $RQ = \{R_{in}, R_{out}\}$
**Output:** A list of services that hold $D_{in} \subseteq R_{in}$ and $D_{out} \supseteq R_{out}$

```
1:  // step1: create and load index
2:  if index exists then
3:     index = loadIndex();
4:  else
5:     index = createIndex();
6:  end if
7:  // step2: match output attributes
8:  candidates = null;
9:  for each attribute d in R_out do
10:    slist = index.find(d);
11:    if candidates == null then
12:       candidates = slist;
13:    else
14:       candidates = candidates ∩ slist
15:    end if
16: end for
17: // step3: match input attributes
18: result = ∅;
19: for each service s in candidates do
20:    if D_in(s) ⊆ R_in then
21:       result.add(s);
22:    end if
23: end for
24: return result;
```

### 3.4 Discussion

Using the composition algorithm introduced above, we can quickly construct a service chain that can fulfill the user request. However, we think there are still many problems to put the algorithm into practical use. We will discuss two



The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 © 2007 IEEE

**Algorithm 2** Algorithm for Service Composition

**Input:** The request $RQ = \{R_{in}, R_{out}\}$
**Output:** All the service chains that can satisfy the request

```
1: for each s in S do // S is the set of all the services
2:    s.isVisited = false; // to avoid infinite loop
3: end for
4: // find services producing R_out \ R_in
5: slist1 = getServices(R_out, R_in);
6: while not completed do
7:    completed = true;
8:    slist2 = ∅;
9:    for each s in slist1 do
10:      if not s.isVisited then
11:         s.isVisited = true; // set the flag
12:         // find services producing D_in(s) \ R_in
13:         slist3 = getServices(D_in(s), R_in);
14:         if slist3 is not empty then
15:            // some new predecessors are founded,
16:            // so it is not the right time to quit.
17:            completed = false;
18:            slist2 = slist2 ∪ slist3;
19:            for each ss in slist3 do
20:               ss.successor.add(s);
21:            end for
22:         end if
23:      end if
24:   end for
25:   slist1 = slist2; // begin next loop
26: end while
27: for each s in S do
28:   if s.isVisted && D_in(s) ⊆ R_in then
29:      // found a chain header
30:      // trace the chain with the recorded successors recursively
31:      traceChain(s);
32:   end if
33: end for
```

fundamental problems here.

The first problem is that the functional semantics of services are completely ignored in such a document-driven composition method. It is possible that two services have the same I/O, while the functional semantics and side-effects of these two services are totally different. For example, both service A and service B consume two integers and produce another integer, while the output of service A is the sum, and the output of service B is the product. As a result, the execution of the constructed service chain is very likely to generate an unexpected result. Formalization of functional semantics is a hard problem. It is a still big challenge in the research of automated service composition and, more generally, automated software engineering.

The second problem is that the composition algorithm does not support flow control, one of the most important aspects of service composition. This limits the applicable area of the algorithm, as most of the real world applications need control structures like if-then, do-until and etc. Without control structures, it is even impossible to express a simple business logic such as if the return message indicates an error, then stop and report the error.

Anyway, in spite of the problems discuss above, the quick composition algorithm can be applied in some specific applications. And the acceleration technologies can be used to speed up the applications in service-oriented architecture (SOA).

## 4 Conclusion

In this paper, we proposed a fast composition method to solve the semantic composition problem defined by WS-Challenge. By making use of the indexing technology, we significantly improved the speed of service lookup, which is a key factor to the efficiency of the composition algorithm.

We notice that indexing is very important for services discovery and composition. In practical applications, requests for retrieving services might be frequent, and indexing can significantly reduce the response time.

## References


[1] M. Aiello, C. Platzer, F. Rosenberg, H. Tran, M. Vasko, and S. Dustdar. Web service indexing for efficient retrieval and composition. In *CEC/EEE'06*, page 63, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[2] Q. A. Liang and S. Y. W. Su. AND/OR graph and search algorithm for discovering composite web services. *International Journal of Web Services Research*, 2(4):48 – 67, 2005.

[3] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE INTERNET COMPUTING*, 8(6):51–59, 2004.

[4] S.-C. Oh, H. Kil, D. Lee, and S. R. T. Kumara. Algorithms for web services discovery and composition based on syntactic and semantic service descriptions. In *CEC/EEE'06*, page 66, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[5] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):24–28, 2003.

[6] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *the Eleventh International World Wide Web Conference*, Honolulu, HI, 2002.

[7] B. Xu, T. Li, Z. Gu, and G. Wu. SWSDS: Quick web service discovery and composition in SEWSIP. In *CEC/EEE'06*, page 71, Los Alamitos, CA, USA, 2006. IEEE Computer Society.





The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services(CEC-EEE 2007)
0-7695-2913-5/07 $25.00 © 2007 IEEE