QoS-Aware Semantic Web Service Composition for SOAs

Xin Yu\textsuperscript{a}, Thomas Weise\textsuperscript{a}, Ke Tang\textsuperscript{a}, and Steffen Bleul\textsuperscript{b}

\textsuperscript{a}Nature Inspired Computation and Applications Laboratory, School of Computer Science and Technology, University of Science and Technology of China
\textsuperscript{b}tetralog systems AG, Munich, Germany

Email: hughys@mail.ustc.edu.cn, \{tweise|ketang\}@ustc.edu.cn, stbleul@gmx.de

Abstract

QoS-aware semantic web service composition concerns finding services from a repository to accomplish a specified task while meeting the Quality of Service (QoS) demands. The composition task is defined in form of a composition request which contains a set of available input parameters and wanted output parameters. If the input parameters given in the request are provided, the services of this set can be executed. Concepts from an ontology describing their semantics are passed to the composition engine. The composer works on a repository of services which posses QoS features. The parameters of these services are semantically annotated in the same way as the parameters in the request. The composer then finds a set of services fulfilling the request. In this paper, we introduce our improved composition system with which we will take part in the Web Service Challenge 2010.

1 Introduction

Nowadays, a corporate software must be built with the anticipation of changes and updates [1]. This is because the software is working in an ever changing environment. For instance, the software of an enterprise is required to be capable of adapting to various changes in the business processes such as accounting, billing, the workflows, and even in the office software. Generally, to re-start the software whenever a change occurs would not be a wise choice unless a new solution can be found efficiently once the new request comes, since some information in the past could be re-used to help with the adaptation [2, 3]. To build such efficient systems, an ideal architecture may be a Service-Oriented Architecture (SOA) [4, 5]. It enables us to modularize the business logic and to implement it in the form of services which can be accessible in a network. Here services build up service processes which represent the workflows of an enterprise. They can be added, deleted, and updated at runtime without interrupting the ongoing business. A SOA can be regarded as a complex system with manifold services as well as \(n:m\) dependencies between services and applications, i.e.:

1. An application may need various service functionalities.
2. Different applications may need the same service functionality.
3. A certain functionality may be provided by multiple services.

Therefore, it is very often that only one service cannot fulfill a complex functionality request. As an alternative, to efficiently compose several services for the requested functionality becomes a necessity. In addition, the final solution of a composition is also required to satisfy some QoS conditions, e.g., to be more stable, cheaper, or faster.

To manage the dependencies above, self-organizing approaches based on a combination of syntactic and semantic service descriptions, to decide whether a service provides a functionality being sought or not, are used. Common syntactic definitions like WSDL specify the order and types of service parameters and return values. Semantic interface description languages like OWL-S [6] or WSMO [7] annotate these parameters with a meaning. While WSDL defines a parameter \texttt{mycar} of the type \texttt{String}, with OWL-S \texttt{mycar} expecting a \texttt{String} actually with a \texttt{Car} can be defined. Via a taxonomy we can now deduce that values annotated as either \texttt{BMW-X5} or \texttt{Benz-S600} can be passed to this service.

A wanted functionality is defined by a set of required output and available input parameters. A service offers this functionality if it can be executed with the given input parameters and its return values contain the needed output values. Many service management approaches utilize semantic service discovery [8]. Still, there is a substantial lack of research on algorithms and system design for fast response service discovery. This is especially the case when combinations of services, the so-called compositions, must be discovered under some QoS constraints.

In this paper we present a modified algorithm based on our previous work with which we successfully took part in
the WS-Challenge 2008 [9, 10]. We adapted our composi-
tion system so it now is compatible with the WS-Challenges
is outlined as follows: Section 2 briefly describes the chal-
lenge rules. The definition of semantic service composition
in the context of QoS is formally given in Section 3. Section
4 illustrates the architecture of the designed system,
followed by the new algorithm with the proposed heuristic
in Section 5. Finally, Section 6 concludes the paper.

2 Challenge Description

Web Services with a semantic description of the inter-
face and QoS characteristics are adopted in the challenge.
The task is to find a composition of services that produces
a set of queried output parameters from a set of given input
parameters, and at the same time optimizes the QoS.

Each challenge contains the following elements: (1) A
Web Service Description Language (WSDL) file containing
a set of services interface descriptions. The number of ser-
cvices can be greatly different between two challenges. (2)
A Web Ontology Language (OWL) file containing a taxon-
omy of concepts relating to the classes of the WSDL files.
(3) A Web Service Level Agreements (WSLA) [14] file con-
taining the QoS non-functional properties for each service
including response time and throughput. (4) One or more
queries to the system to be run. Every query is a set of input
parameters and output parameters that generally require a
composition of services to be satisfied.

The format of the output solution is in a Business Pro-
cess Execution Language (BPEL) schema. It contains the
service composition with the lowest response time and the
service composition with the highest throughput. Response
time measures the interval between the time a request is re-
sented, by a Web service and the time a reply to the request
is sent. It is expressed in time units. The throughput mea-
sures the number of requests that a web service can handle
within a time unit.

In the challenge, there are 4 types of BPEL activities:
Sequence, Case, Flow, and Invoke. The QoS of an invo-
cation is the same as that of the invoked service. For a
sequence consisting of \( A_1, A_2, \ldots, A_n \), a flow \( B \) con-
sisting of \( B_1, B_2, \ldots, B_n \), and a case \( C \) consisting of
\( C_1, C_2, \ldots, C_n \), the response time \( R \) and throughput \( T \)
are calculated as follows:

\[
R(A) = \sum_{i=0}^{n} R(A_i) \quad (1)
\]

\[
R(B) = \max(R(B_1), R(B_2), \ldots, R(B_n)) \quad (2)
\]

\[
R(C) = \min(R(C_1), R(C_2), \ldots, R(C_n)) \quad (3)
\]

\[
T(A) = \min(T(A_1), T(A_2), \ldots, T(A_n)) \quad (4)
\]

\[
T(B) = \min(T(B_1), T(B_2), \ldots, T(B_n)) \quad (5)
\]

\[
T(C) = \max(T(C_1), T(C_2), \ldots, T(C_n)) \quad (6)
\]

3 Semantic Service Composition

To facilitate the discussion of the idea, we introduce
some prerequisites first. Assume that all concepts in the
knowledge base are members of the set \( M \) and can be rep-
resented as nodes in a wood of taxonomy trees.

**Definition 1 (subsumes)** Two concepts \( A, B \in M \) can be
related in one of four possible ways. We define the predicate
\( \text{subsumes} : (M \times M) \rightarrow \{\text{true}, \text{false}\} \) to express this
relation as follows:

1. \( \text{subsumes}(A, B) \) holds if and only if \( A \) is a general-
ization of \( B \) (\( B \) is then a specialization of \( A \)).
2. \( \text{subsumes}(B, A) \) holds if and only if \( A \) is a special-
ization of \( B \) (\( B \) is then a generalization of \( A \)).
3. If neither \( \text{subsumes}(A, B) \) nor \( \text{subsumes}(B, A) \) holds,
   \( A \) and \( B \) are not related to each other.
4. \( \text{subsumes}(A, B) \) and \( \text{subsumes}(B, A) \) is true if and
   only if \( A = B \).

The subsumes relation is transitive, and so are generaliza-
tion and specialization.

If a parameter \( x \) of a service is annotated with \( A \) and a value
\( y \) annotated with \( B \) is available, we can set \( x = y \) and
call the service only if \( \text{subsumes}(A, B) \) holds (contravari-
ance). This means that \( x \) expects less or equal information
than given in \( y \).

The set \( S \) contains all the services \( s \) known to the registry.
Each service \( s \in S \) has a set of required input concepts
\( s.in \subseteq M \) and a set of output concepts \( s.out \subseteq M \) which
it will deliver on return. We can trigger a service if we can
provide all of its input parameters.

For a composition request \( R \), it consists of a set of avail-
able input concepts \( R.in \subseteq M \) and requested out-
put concepts \( R.out \subseteq M \). A composition algorithm dis-
covers a (topologically sorted) set of \( n \) services \( S = \{s_1, s_2, \ldots, s_n\} : s_1, \ldots, s_n \in S \).
As shown in Equation 7, the first service \( s_1 \) of a valid composition can be executed
with instances of the input concepts \( R.in \). Together with
\( R.in \), its outputs \( s_1.out \) are available for executing the
next service \( s_2 \) in \( S \), and so on. The composition provides
outputs that are either annotated with exactly the requested
concepts \( R.out \) with more specific ones (covariance).

For each composition solving the request \( R \), isGoal\( (S) \) will hold:

\[
\text{isGoal}(S) \Leftrightarrow \forall A \in s_i.in \ \exists B \in R.in : \text{subsumes}(A, B) \land \forall A \in s_i.in, i \in \{2..n\}
\]

\[
\exists B \in R.in \cup s_{i-1}.out \cup \ldots \cup s_1.out : \text{subsumes}(A, B) \land \forall A \in R.out \ \exists B \in s_1.out \cup \ldots \cup s_n.out \cup R.in : \text{subsumes}(A, B) \cup \text{fulfillsQoS}(S).
\]

(7)

The search space that needs to be investigated in web ser-
vice composition [15] is the set of all possible permutations
of all possible sets of services. In order to proceed in this space, we define the operation promising which obtains the set of all services \( s \in S \) that produce an output parameter annotated with the concept \( A \) (regardless of their inputs).

\[
\forall s \in \text{promising}(A) \ \exists B \in s.\text{out} : \text{subsumes}(A, B) \quad (8)
\]

We define the number known \( S \) of know concepts as

\[
\text{known}(S) = R.\text{in} \cup \forall s \in S \ s.\text{out} \quad (\text{and set wanted}(S) \text{of unsatisfied parameters as } \forall A \in \text{wanted}(S) \Rightarrow \exists s \in S : A \in s.\text{in} \wedge A \notin \text{known}(S)).
\]

4 Composition System Overview

An application accesses the composition system illustrated in Figure 1 by submitting a service request through its Web Service Interface and also provides the service descriptions and their semantic annotations in form of WSDL and XSD formatted files or OWL-S descriptions. These documents and their semantic annotations in form of WSDL and XSD formatted files or OWL-S descriptions are parsed by a fast SAX formatted Input Parser. These descriptions and their semantic annotations in form of WSDL and XSD formatted files or OWL-S descriptions are also provided to a fast SAX-based Input Parser. The composition process itself is started by the Strategy Planer which allocates the system resources and chooses the composition algorithm that seems to be most appropriate for the given problem. In addition to in the original composition algorithm of our system, we also applied the devised heuristic function to a memetic algorithm [16, 17].

The software modules encapsulating the two composition algorithms have direct access to the Knowledge Base and to the Service Register. Although every algorithm and composition strategy is unique, they all work on the same data structures. One or more composition algorithms solve the composition challenge and pass the solution to a SAX-based Output Writer, a very fast XML output generator. The resulting document is then returned through the Web Service Interface. The knowledge base and the service registry are constituted by instances of the classes Concept and Service, as illustrated in Figure 2. Instances of Concept provide the method promising which corresponds to the predicate promising introduced in Equation 8. In order to determine its result, all the specializations of the concept have to be traversed and their promising services have to be accumulated. The crux of the routine is that this costly traversal is only performed once per concept. Our experiments have shown that the resource memory is not a bottleneck even for largest service repositories. Hence, promising caches its results. This caching is done in a way that is thread-safe on one hand and does not need any synchronization. The same holds for all features of the knowledge base: multiple algorithms may run in parallel, using the same knowledge base without any synchronization-induced delay, race conditions or inconsistencies.

5 QoS-Aware Semantic Composition

In our past experiences, heuristic searches are the best performing approaches for semantic web service composition. In 1, a heuristic \( c \) helps to decide which nodes are to be expanded next. As main composition method we therefore define a greedy algorithm that internally selects the best currently known candidate composition according to a heuristic in form of a comparator function \( c : S^0 \rightarrow \mathbb{R} \). The comparator function \( c(S_1, S_2) \) will be below zero if \( S_1 \) seems to be closer to the solution than \( S_2 \) and greater than zero if \( S_2 \) is a more prospective candidate. Algorithm 1 is also used as local search in our Memetic Algorithm approach. We can easily derive suitable comparator functions for web service composition. So far, we do not have knowledge of
any heuristic with better behavior than the function $c_{comp}$ that we present in Algorithm 2.

**Algorithm 2: $r = c_{comp}(S_1, S_2)$**

<table>
<thead>
<tr>
<th>Input: $S_1, S_2$ two composition candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: $r \in \mathbb{Z}$ indicating whether $S_1$ ($r &lt; 0$) or $S_2$ ($r &gt; 0$) should be expanded next</td>
</tr>
</tbody>
</table>

```plaintext
begin
  $i_1 \leftarrow \text{wanted}(S_1)$;
  $i_2 \leftarrow \text{wanted}(S_2)$;
  if $i_1 \neq i_2$ then return $i_1 - i_2$
  if $R(S_1) \neq R(S_2)$ (or $T(S_1) \neq T(S_2)$) then return $R(S_1) - R(S_2)$ (or $T(S_2) - T(S_1)$);
  return $c_{2008}(S_1, S_2)$ (see [15, 9])
end
```

### 6 Conclusions

In this technical description, we formally defined the problem of semantic web service composition. Based on this definition, we derived a highly efficient composition algorithm according to a QoS-based heuristic function. We also applied this heuristic function to in a Memetic Algorithm, and employed a planner to allocate computation resources the two algorithms. None of the techniques tested in our experiments could outperform this combination. We have been adapted our algorithm and look forward to the Web Service Composition 2010.

### Acknowledgements

This work was supported in part by the National Natural Science Foundation of China under Grant U0835002.

### References


