An Infrastructure for Evolving Dynamic Web Services Composition

Frederico G. Alvares de Oliveira Jr.
Technological Institute of Aeronautics (ITA)
Prac¸a Marechal Eduardo Gomes, 50
Sao Jose dos Campos - SP - Brasil

José M. Parente de Oliveira
Technological Institute of Aeronautics (ITA)
Praça Marechal Eduardo Gomes, 50
S˜ao Jose´ dos Campos - SP - Brasil

Abstract

Web services are becoming de facto a standard for data exchanging, since they provide a clear way to express and access information throughout the Internet. Sometimes, one service alone is not able to perform a certain task, and it might be necessary to compose two or more services in order to accomplish that task. In this context, semantic web services play an essential role, since they provide a framework to formally describe services in a way that it is possible to machines to automatically discover, compose, invoke, and monitor them. On the other hand, only functionality is not enough to fulfill users' satisfaction, in the sense that service providers should guarantee the quality delivered by their services. In this paper, we present an infrastructure for dynamic web service composition. The main element of the infrastructure is the composition mechanism which is based on quality of service. Experimental results indicates the proposal soundness.

1 Introduction

A web service can be defined as a piece of software that conforms to a set of open interoperability facilities, such as WSDL (Web Service Description Language) [7], SOAP (Simple Object Access Protocol) [4], and UDDI (Universal Description, Discover and Integration) [14], for description, messaging protocol and discovering, respectively [12]. These facilities allow the integration of systems written in different languages and running on computers with different platforms. However, all of them offer only a common grammar for describing, publishing and exchanging information of web service and workflows, which means that there is a lack of semantics in the information provided by services defined with those facilities [2].

A Semantic Web Service (SWS) is a service whose input, output, preconditions and effects (IOPEs) are associated with a formal description rather than just a datatype. A straightforward impact of having Web Services semantically annotated is the fact that their functionality can be automatically matched (discovered) and composed with functionalities provided by other services when only one service is not able to produce the desired functionality. Consequently, new functionalities and applications can be dynamically built based on existing services.

Many times, however, only the matching of service functionalities based on input and output description is not enough to fulfill the users requirements. A service user, i. e. a person or another computer, may impose constraints on the required service. For this purpose, services should guarantee a minimum level of quality to be delivered to their consumers. The quality of a service is often expressed by means of non-functional attributes, such as performance, availability, cost etc.

This paper proposes an infrastructure that supports the dynamic composition of SWS, in which every new generated composition is made readly available in order to be used as a new component of future compositions. This infrastructure comprises two distinct modules: the composer, which is in charge of composing the services before executing them; and execution engine, which manages the actual execution. For the latter we leverage existing work [13], while we concentrate our contributions on the composer module.

The remainder of this paper is organized as follows: Section 2 presents the related works; Section 3 presents the proposed infrastructure; Section 4 shows some experimental results and a motivating example of usage of the proposed infrastructure and Section 5 gives the final remarks and future works.

2 Related Work

In this section, we discuss a selected set of recent relevant works related to our proposal.

In [20], a middleware for QoS-based web service composition was proposed. It finds the optimal set of service

implementations for a static abstract service composition. Similarly, in [5], an approach for runtime adaptation of QoS-based web service composition was proposed. In our work, we consider that there is no information about the structure of the composition, and the services are discovered and composed on demand.

In [6], Chafle et. al. proposed an approach for adaptation of service composition that considers QoS properties. The adaptation is performed by a monitoring system that triggers adaptation procedures in different stages (runtime, physical, and logical). However, it was not clear what is the overhead of that spread of information on every environment changing.

Claro et. Al [8] proposed a framework for semantic web service composition that relies on GPA (Goal-oriented Planning Algorithm), and allows re-planning of compositions. In addition, the composition is based upon user profiles, which besides grouping personal information (name, contact, etc.), non-functional information (Cost, Turnover, Execution Time and Reputation) are considered. The profile is saved as a predicate along with a user rate on the service. Contrary to our approach, the framework provides all possible compositions before ranking them. As our concern is to provide a correct solution spending as less processing time as possible, our algorithm does not search in all the state space.

Oh et al. [15] proposed an algorithm for semantic and syntactic service composition. Similarly to the algorithm of our infrastructure, their algorithm takes into account the overall cost of invoking a service in order to improve the quality of compositions. However, their approach is limited to the composition algorithm, and nothing else about the composition infrastructure is presented.

In [17], three different approaches for service composition were proposed. An uninformed approach, which is based on a Depth-First Search, an informed approach, which is based on a heuristic that guides the composition process, and a generic approach, which is based on evolutionary algorithms. Experiments have shown that the heuristic-based approach outperformed the others. In fact, our approach was developed based on that informed approach. However, we add to that algorithm QoS properties and propose a whole evolving dynamic service composition infrastructure.

Yan et al. [18] proposed an approach for automated semantic service composition based on And/Or Graph. Although experiments have shown that the proposed algorithm performs well, specially in cases of parallel composition of web services, quality issues are not discussed.

From the literature review carried out, it is possible to infer two important aspects. First, the use of QoS for the dynamic composition of Semantic Web services together with a fast search algorithm is not used. Second, successful compositions based on QoS that are generated are not made promptly available for new compositions. These two aspects not used together in other works can provide important improvements in terms of general QoS of compositions, as well as reductions in their execution times.

3 An Infrastructure for Dynamic Web Services Composition

This section describes the infrastructure porposed in this paper. Firstly, an overview of the infrastrucre will be given, and then the module composer will be described in more details along with its implementation.

3.1 Infrastructure Overview

Figure 1 depicts an overview of the infrastructure proposed in this paper. This overview represents a three-layered architecture, from the most abstract (Semantics) to the most concrete (Implementation). A composition starts when a Requester, a person or a computer system, requests some functionality (required outputs) by providing some known information (provided inputs) and imposing Quality of Service (QoS) Constraints for the Composer which, by its turn, is the module responsible for finding a set of services that together provide the required outputs and meet the QoS constraints. Services are described by a Service Ontology, which can be described in OWL-S [1] or WSMO (Web Service Modeling Ontology) [2], for instance, and their functionality are mapped onto concepts defined by the Domain Ontology, which can be described in OWL (Web Ontology Language) [3], for instance. In addition, Quality of Service Characteristics (or criteria) provided by the services are also specified in SLA (Service Level Agreement) in OWL-S or even in a simple XML file. The composition algorithm tries to match one or more services by reasoning on concepts that describe the service functionality and which definition is provided by the Domain Ontology. Furthermore, the composition should be aware if the composition that is being formed also meet the QoS Constraints.

Once a composition is found, it can be represented by a Composite Service in OWL-S in order to be executed by the Execution Engine. The Execution Engine's main responsibility is to invoke constituent services of a composition by obtaining their syntactical description (WSDL, for instance), which has information about their location and how to invoke them. The implementation of the services to be invoked might be available anywhere over the Internet. Apart from that, the Execution Engine is in charge of monitoring the execution in order to keep QoS values up to date,

[1] http://www.w3.org/Submission/OWL-S/
[2] http://www.w3.org/Submission/WSMO/
[3] http://www.w3.org/TR/owl-features/

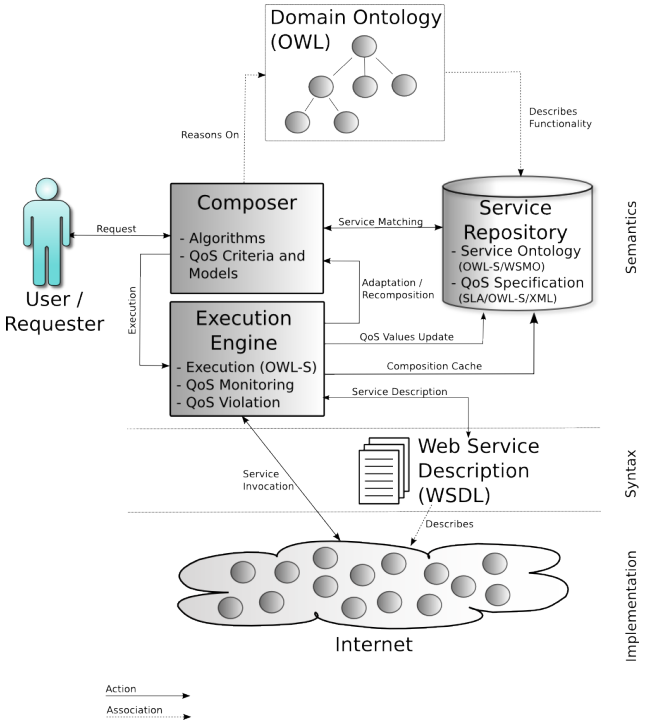

Figure 1: Proposed Infrastructure

as well as to detect QoS violation i.e. if a service does not meet the QoS previously announced.

It is important to mention that in some QoS criteria (e. g. availability, duration etc), the values assigned to them vary according to the number of execution and thus every execution should be monitored. In other words, it is also the execution engine's responsibility to observe the quality delivered by each executed service as well as to update their corresponding specification. For example, the quality criterion Duration is measured based on the average of the past execution observations. Hence, it may be established for instance that in every 100 executions of a certain service, its Duration should be updated.

If the execution of a composition fails, the Execution Engine can request the composer an adaptation of the failed composition or even another composition. If everything goes well, the executed composition is stored as a new service in the Service Repository, avoiding all the search procedure when the same request arrives later on.

Technologies such as OWL, OWL-S, WSMO were mentioned only to show some technologies that can be used to deploy the solutions for Evolving Dynamic Web Service Composition, and thus, other similar technologies could be used instead.

3.2 Composer Model

Semantic Web Service Composition consists in composing two or more well-described (ontology-based) services in order to accomplish a requested functionality. The proposed approach in this work is intended to perform Dynamic Service Composition taking into account both the semantic description of a service and its non-functional properties that composes the quality it delivers. The algorithm to perform this composition receives as input a request, which consists of the provided input concepts, required output concepts and QoS constraints, and produces as output a set of services that together can provide the required concepts specified in the request, as illustrated in Figure 2.

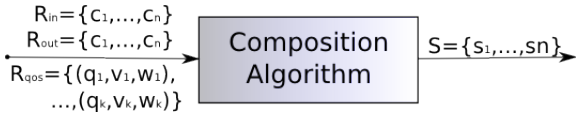

Figure 2: Proposal Overview.

Each concept $(c_1, c_2, \ldots, c_n)$ in the request input or output is defined in a ontology named Domain Ontology. Each QoS constraint consists of a triple with the quality criterion, a value representing a constraint on this criterion and a weight representing the user's preference on this criterion. Following are some definitions extended from [17] to incorporate QoS.

Domain Ontology. A domain ontology $\mathcal{O}$ consists of a set of concepts used to describe the domain in which a group of services is inserted. These concepts are related to each other according to subsumption relations, i. e. a concept $c_1 \in \mathcal{O}$ subsumes another concept $c_2 \in \mathcal{O}$ if c_1 is a superclass of c_2; c_2 subsumes c_1 if c_1 is a subclass of c_2; c_1 and c_2 subsume each other if they are the same concept.

Request. A composition request $\mathcal{R}$ consists of a set of provided inputs $\mathcal{R}_{in} \subseteq \mathcal{O}$, a set of required outputs $\mathcal{R}_{out} \subseteq \mathcal{O}$ and a set of quality of service constraints $\mathcal{R}_{qos} = \{(q_1, v_1, w_1), (q_2, v_2, w_2), \ldots, (q_k, v_k, w_k)\}$, where q_i $(i = 1, 2, \ldots, k)$ is a quality criterion, v_i is the required value for criterion q_i, w_i is the weight assigned to this criterion such that $\sum_{i=1}^{k} w_i = 1$, and k the number of quality criteria involved in the request. A quality criterion can be either negative, i. e. the higher the value the lower the quality, or positive, i. e. the higher the value the higher the quality.

A service composition is built by discovering services available in a Service Repository that match with concepts required by either service inputs or request output. A service is selected to take part of certain composition if at least one of its output concepts is related to a required concept

(request output or a required input of another service in the composition) according to the subsumption relations above mentioned.

A valid composition is a composition where all the required output concepts can be provided by its constituent services as well as each service input concepts should be provided by either the available concepts in the request or other services in the composition. In addition, a valid composition also must meet the imposed QoS constraints.

Service Repository. A Service Repository $\mathcal{D}$ consists of a set of available services. Each service $s \in \mathcal{D}$ is composed of a set of required inputs $s_{in} \subseteq \mathcal{O}$, a set of provided outputs $s_{out} \subseteq \mathcal{O}$ and a set of provided quality criteria $s_{qos} = \{(q_1, v_1), (q_2, v_2), \ldots, (q_k, v_k)\}$, where q_i $(i = 1, 2, \ldots, k)$ is a quality criterion, v_i is the value associated to criterion q_i provided by the service, and k is the number of criteria involved.

Service Composition. A Service Composition $\mathcal{C}$ is a Direct Acyclic Graph (DAG) with vertices $\mathcal{C}_v = \{s \mid s \in \mathcal{D}\}$ and edges $\mathcal{C}_e = \{(u, v) \mid u, v \in \mathcal{C}_v \wedge \exists c_1 \in u_{out} \wedge \exists c_2 \in v_{in} : c_2 \text{ subsumes } c_1\}$

Valid Composition. A composition $\mathcal{C}$ is considered valid for a request $\mathcal{R}$ if the predicate $valid(\mathcal{C}, \mathcal{R})$ in Equation 1, holds.

$$
\begin{aligned}
valid \quad & (\mathcal{C}, \mathcal{R}) \Leftrightarrow \\
& \forall c_1 \in (\mathcal{R}_{out} \bigcup\nolimits_{\forall s \in \mathcal{C}_v} s_{in}) \; \exists c_2 \in (\mathcal{R}_{in} \bigcup\nolimits_{\forall s \in \mathcal{C}_v} s_{out}) \\
& (c_1 \text{ subsumes } c_2 \wedge \\
& \quad \forall (q, v, w) \in \mathcal{R}_{qos} \\
& \quad (q \text{ is negative } \wedge overall(\mathcal{C}, q) \leq v) \vee \\
& \quad (q \text{ is positive } \wedge overall(\mathcal{C}, q) \geq v)))
\end{aligned} \tag{1}
$$

where $overall(\mathcal{C}, q)$ represents the model that calculates the overall value of criterion q in composition $\mathcal{C}$.

The following subsections gives an overview of the Greedy Search strategy used to construct the algorithm which is the basis of the proposed approach. In addition, the adaptation made in the QoS models for composite service is described.

3.2.1 Quality of Service Models

Quality of Service (QoS) is the quality delivered by one service, expressed by means of non-functional characteristics with quantifiable parameters [16]. Examples of those criteria are described as follows:

- **Duration** ($q_{du}(s)$) of a service s measures the expected delay between the moment when a request is sent and the moment when the result are received;
- **Reputation** ($q_{rep}(s)$) of a service s is a measure of its trustworthiness. It is given by end users' opinions on the service;
- **Availability** ($q_{av}(s)$) of a service s is the probability that it is accessible and
- **Price** ($q_{pr}(s)$) of a service s is the fee that its invokers have to pay for invoking it.

In order to provide the QoS models of a composite service $S = \{s_1, s_2, \ldots, s_n\}$, Zeng et al. [20] defined a set of aggregation functions, which are presented as follows.

The equation 2 defines the overall price of a composite service by summing all elementary service's price.

$$q_{pr}(S) = \sum_{i=1}^{n} q_{pr}(s_i) \tag{2}$$

Since a composition can be both sequential or parallel, the overall duration is computed by applying the Critical Path Algorithm in order to always capture the execution duration of the most time-demanding service in a parallel composition, as expressed in 3.

$$q_{du}(S) = CPA(S, q_{du}) \tag{3}$$

The overall reputation is the average of the reputation of all constituent services of the composition, as expressed in Equation 4.

$$q_{rep}(S) = \frac{1}{n} \sum_{i=1}^{n} q_{rep}(s_i) \tag{4}$$

The aggregation function for availability is defined by Zeng et al. [20] as the probability of all events happen together. So, it is computed by the product of all probabilities, as defined in Equation 5.

$$q_{av}(S) = \prod_{i=1}^{n} (q_{av}(s_i))^{z^i} \tag{5}$$

where z^i is 0 if service s_i is part of a critical path and 1 otherwise. It means that if a service s_i is not available, the execution engine have some time (since s_i is not in a critical path) to replace s_i for another service.

3.2.2 The Algorithm

This subsection presents an extension of the algorithm proposed in [11]. The Algorithm 1 performs a service composition for a given request $\mathcal{R}$. It starts by finding all services that can provide as output a concept which is semantically equivalent (by subsumption reasoning) to the required outputs specified in $\mathcal{R}_{out} \in \mathcal{R}$ (Algorithm 1, lines 2-9). A service will be a candidate solution if it can meet the QoS constraints specified in $\mathcal{R}_{qos} \in \mathcal{R}$. The predicate

$meetQoS(\mathcal{R}, \mathcal{C})$ (Algorithm 1, lines 5-7) filters the candidate compositions that no longer meet the constraints imposed in the request and therefore should be discarded. The Equation 1 cannot be used since in some criteria, such as Reputation, it is not possible to determine if a certain composition does not meet the constraints imposed by the request without knowing the final number of services in the composition, because its value is determined by the average of the services' individual values. In such cases, this kind of criterion is just ignored and is not checked.

The candidate compositions are sorted according to a comparator function which compares two candidate compositions $\mathcal{C}_1$ and $\mathcal{C}_2$ and return a value below zero if $\mathcal{C}_2$ is a more prospective composition than $\mathcal{C}_1$, above zero if $\mathcal{C}_1$ is a more prospective composition than $\mathcal{C}_2$, and zero if both are equally evaluated (Algorithm 1, line 11). The comparator function proposed by [17] considers four composition properties: (i) $known$ concepts; (ii) $unknown$ concepts; (iii) $eliminated$ concepts; and the number of services in a composition. The property $known$ (Equation 6) of a given composition $\mathcal{C}$ and request $\mathcal{R}$ is the set of all known concepts, i. e. the input concepts provided by the requester and output concepts provided by the output of all services in composition $\mathcal{C}$. The property $unknown$ (Equation 7) of a given composition $\mathcal{C}$ is a set of the concepts needed to make the composition $\mathcal{C}$ (functionally) valid. This set is composed of the concepts required in the request (i.e. request outputs) and concepts required to execute each service in the composition $\mathcal{C}$. The property $eliminated$ (Equation 8) is a set of unknown concepts that has already been provided.

$$known(\mathcal{C}) = \mathcal{R}_{in} \bigcup_{\forall s \in \mathcal{C}_v} s_{out} \tag{6}$$

$$unknown(\mathcal{C}) = \{ \quad c \mid \exists s \in \mathcal{C}_v : c \in s_{in} \wedge c \notin known(\mathcal{C}) \} \tag{7}$$

$$eliminated(\mathcal{C}) = \{ \quad c \mid \exists s \in \mathcal{C}_v : c \in s_{in} \wedge c \in known(\mathcal{C}) \} \tag{8}$$

In this work, a fifth property was added: The Overall Quality of Service Score. This property aims at representing in a single value all the quality criteria values of a composition. For this purpose, the Multiple Criteria Decision Making (MCDM) [19] named Simple Additive Weighting (SAW) [19] technique is used. This technique allows the calculation of a score taking into account several, and sometimes conflicting, criteria. First, all criteria are put in the same scale. Equations 9 and 10 provides a function that calculates the scaled value of a criterion q of composition $\mathcal{C}$ considering if q is positive (i.e. the higher the value the higher the quality) or negative (i.e. the higher the value the lower the quality). In practice, the values are scaled based

Input: $\mathcal{R}$ - the user Request, $\mathcal{D}$ - the Service Repository
Result: $\mathcal{C}$ - the composition found or $\emptyset$
Data: X - the set of candidate compositions

```
1  begin
2      foreach outR ∈ R_out do
3          foreach s ∈ D | ∃ c ∈ s_out(outR subsumes c) do
4              C_v ⟵ {s};
5              if meetQoS(R, C) then
6                  append(X, C);
7              end
8          end
9      end
10     while X ≠ ∅ do
11         sort(X, comparator);
12         C ⟵ removeMostProspective(X);
13         if valid(C, R) then
14             return C;
15         end
16         foreach outR ∈ unknown(C) do
17             if ¬∃s ∈ D(∃ c ∈ s_out(outR subsumes c)) then
18                 break;
19             end
20             foreach s ∈ D | ∃ c ∈ s_out(outR subsumes c) do
21                 newCandidate_v ⟵ C_v ∪ {s};
22                 newCandidate_e ⟵ C_e ∪_{∀s2∈c_v(∃c2∈s2_in c2 subsumes outR)} {(s, s2)};
23                 if meetQoS(R, newCandidate) then
24                     append(X, newCandidate);
25                 end
26             end
27         end
28     end
29     return ∅;
30 end
```

Algorithm 1: Composition Algorithm.

on the highest and lowest values of a given criterion of the compositions in the list of candidates compositions.

$$sc(\mathcal{C}, q) = \begin{cases} \frac{overall(\mathcal{C},q) - q_m(q)}{q_M(q) - q_m(q)} & \text{if } q_M(q) - q_m(q) \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (9)$$

$$sc(\mathcal{C}, q) = \begin{cases} \frac{q_M(q) - overall(\mathcal{C},q)}{q_M(q) - q_m(q)} & \text{if } q_M(q) - q_m(q) \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (10)$$

where $q_M = Max\{overall(\mathcal{C}, q) : \mathcal{C} \in X\}$ and $q_m = Min\{overall(\mathcal{C}, q) : \mathcal{C} \in X\}$ and X is the list of candidate compositions.

The overall QoS value of a composition considering all quality criteria can be determined by a score which takes into account the scaled values and the weights provided in the request $\mathcal{R}$ associated with each criteria. Equation 11 defines the function that calculates the Overall QoS Score for a given composition $\mathcal{C}$.

$$score(\mathcal{C}) = \sum_{\forall(q,v,w) \in \mathcal{R}_{qos}} sc(\mathcal{C}, q) * w \quad (11)$$

Originally, the comparator function choose one of the compositions that has no $unknown$ concepts left. If both compositions does not need any $unknown$ concept left, it returns the composition that has less services. If both of them have $unknown$ concepts, the function returns the composition with more $eliminated$ concepts. But, if both have the same number of $eliminated$ concepts, it returns the composition that has less $unknown$ concepts. If both of them have the same number of $unknown$ concepts, it returns the composition with less services. But if both of them have the same number of services, the function returns the composition with more $known$ concepts.

The comparator function now should also take the Overall QoS Score into account, along with the other four composition properties ($unknown$ concepts, $eliminated$ concepts, composition size, and $known$ concepts). A number of experiments were performed in order to figure out the best place in terms of priority to put the QoS Score property in the comparator function without affecting the algorithm performance. The comparator function works as follows:

1. **unknown = 0 (↑) -** The comparator function returns the composition that has no $unknown$ concepts left, that is, the composition that is functionally valid.

   1.1. **score (↑) -** The Overall QoS Score takes place as a tiebreaker when both compositions have the same number of $unknown$ concepts. Thus, when both compared compositions have no $unknown$ concepts left, the compositions with higher score is chosen.

      1.1.1. **size (↓) -** If both compositions have the same score, the one with less services is chosen.

2. **eliminated (↑) -** The composition with more eliminated concepts is chosen when both compositions still have $unknown$ concepts.

3. **unknown (↓) -** When both compositions have the same number of $eliminated$ concepts, the one with less $unknown$ concepts is chosen.

4. **score (↑) -** The composition with higher score is chosen when both compositions have the same number of $unknown$ concepts.

5. **size (↓) -** When both compositions have the same score, the one with less services is chosen.

6. **known (↑) -** Finally, if both compositions are of the same size, the composition with more $known$ concepts is chosen.

Once the list is sorted using the comparator function, it is possible to choose the composition which is the most prospective composition in terms either of functionality and Quality of Service (Algorithm 1, line 12).

If the chosen composition is valid, that is, if the composition does not need any additional concept to supply the services inputs or request outputs, and meets the constraints imposed in the request, it is returned by the algorithm (Algorithm 1, lines 13-15). Otherwise, the chosen composition is expanded to form new candidate compositions with services that provide concepts required by the composition (Algorithm 1, lines 16-27). Again, if the actual composition does not meet the QoS constraints, it is discarded (Algorithm 1 lines 23-25). If at least one unknown concept cannot be provided with the services in the repository this composition is also discarded (Algorithm 1, lines 17-19). The algorithm runs until a candidate solution is found or all candidate solutions are expanded and rejected, which is the worst case of execution.

If the Overall QoS Score property had preference against the $unknown$ and $eliminated$, the algorithm would perform similarly to a Breadth-first search. For instance, suppose the set of criteria Price, Duration and Availability. The compositions with less services would more likely have higher scores, since it would be cheaper, more available and with less time-demanding. As a consequence, these compositions would be always selected to be expanded. However, as the composition is expanded, it gets larger, decreasing its chance to be chosen in the next time.

On the other hand, by considering the Overall Score as a tiebreaker when properties $eliminated$ and $unknown$ of the compared compositions are equal, the search process will not be affected very much in terms of performance. The advantage then is that it will always expand the composition with higher score.

3.3 Composer Implementation

The implementation of the infrastructure proposed in this paper follows the layered approach depicted in Figure 3.

OWL, OWL-S, QoS. The Domain Ontology is expressed in OWL. The Service Repository is expressed in a set of OWL-S files. For the sake of compatibility, the QoS attributes of each services are expressed separately in XML. Finally, as a result of the composition process, another OWL-S file, describing the composite web service, is generated to be executed by the Execution Engine. If the execution succeeds, the composition is stored in the Service Repository, avoiding process time spent by the the composer when the same request arrives later on.

Jena, OWLS-API, SAX. The Composer makes use of three APIs for parsing the files described on the underlying layer. Jena [4] is used for parsing ontologies described in OWL. OWL-S API [5] is used for parsing Semantic Web Services described in OWL-S. SAX [6] is used to parse XML files used to describe the QoS attributes. Finally, the OWL-S API is used again to generate the OWL-S of the resulting composition.

IO Manager. On top of the APIs described above, the IO Manager layer is responsible for translating the repositories and ontologies stored in OWL, OWL-S, as well as the QoS description, in Business Rules layer classes. In the way back, the IO Manager serves as a bridge to translate the Business Rules classes into OWL-S models.

Business Rules. This layers represents the model (in Java classes) for representing entities such as Request, Concept, Service, and so on. The model is described in the UML (Unified Modeling Language) Class Diagram, illustrated in Figure 4.

Algorithms and Heuristics. The Java implementation of the algorithm and the heuristics presented in subsection 3.2.2.

[4] http://jena.sourceforge.net/
[5] http://owlapi.sourceforge.net/
[6] http://www.saxproject.org/

The layered approach allows to add other algorithms and heuristics than those presented in subsection 3.2.2, as well as to work with other kinds of Repository and Ontology representations.

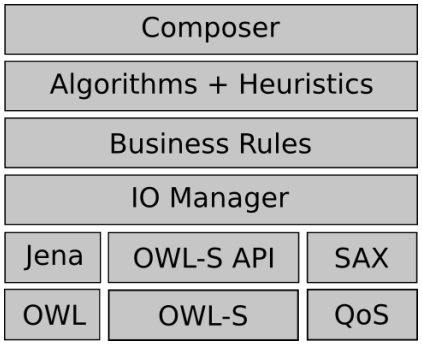

Figure 3: Composer Layered Representation

4 Experiments

This section aims at evaluating the proposed infrastructure, focusing on the composer module. First, some Experimental Results obtained over the Web Service Challenge Benchmark [3, 1] is described, then a Motivating Scenario is presented.

4.1 Experimental Results

The experiments were performed under a variation of test sets provided by the Web Service Challenge (WSC) [3, 1] on a computer with processor Intel Core 2 Duo - 2.16 GHz, with 2GB of RAM. The test sets consist of groups of repositories, ontologies, requests and solutions. For each repository, there is an ontology associated, a set of requests and their respective solutions, as illustrated in Table 1. In WSC, the service specification was limited to required inputs and provided outputs, so Quality of Service was not considered in this competition. Thus, it was necessary to prepare the WSC test sets in order to accommodate quality values for each service presented in each test set. The set of criteria used in these experiments is composed of those defined in subsection 3.2.1, that is, Duration, Price, Availability, Successful Execution Rate and Reputation, with values ranging from 20 to 500, 0.10 to 1.75, 0.3 to 1.0, 0.3 to 1.0 and 0.3 to 1.0, respectively, randomly generated, following a uniform distribution. In addition, as the WSC provides the possible solution for each request, it is possible to have a QoS-based ranking of these possible solutions by exhaustively searching the solutions with highest QoS score (Equation 11). This allows the comparison of compositions found by the algorithm that takes into account QoS, the one that

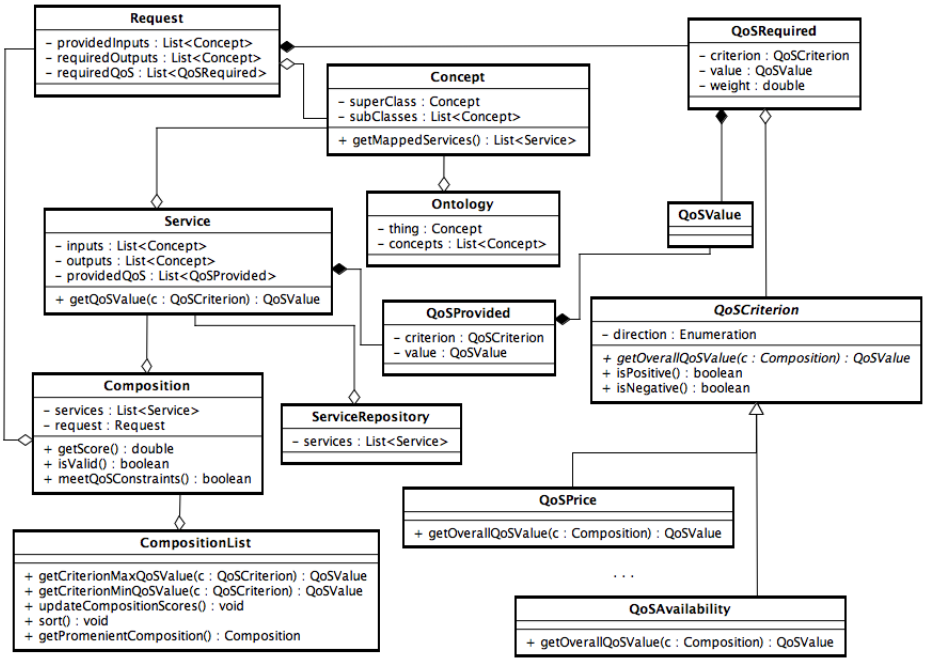

Figure 4: Business Rule Layer Class Diagram

does not take into account QoS, and the optimal solution for each defined request.

All requests were performed using the following weight distribution: Duration: 0.3; Price: 0.3; Availability: 0.1; Reputation: 0.2; Successful Execution Rate: 0.1. The restriction imposed by the user on each quality criteria was omitted in these experiments, since it is just simple control checking performed after a composition is found. As previously stated, these experiments aim at evaluating the proposed approach in terms of feasibility and improvement on the QoS of the compositions.

The experiments compare the compositions found by the approach proposed here and the approach proposed in [17] with regard to the execution time and the overall quality of the compositions. Figure 5 illustrates the overall quality of the compositions found by both approaches for each request of the test set as well as the optimal solution. As can be seen, for all the request the algorithm of the proposed composer could find compositions with better QoS values than those found by the other approach. Figure 6 shows the execution time for finding the composition for each one of the requests for both approaches. Although the compositions found by the proposed composer have better overall QoS values, the time for finding them is almost the same of the other approach.

The experiments developed were intended to evaluate the impacts of the insertion of QoS in the algorithm proposed by [17]. The impacts can be negative, for instance the over-

Table 1: Experimental Test Sets.

$\mathcal{R}$	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{O}\rvert$	$\lvert\mathcal{C}\rvert$	Solutions
1	118	1590	2	9
2	118	1590	3	8
3	118	1590	4	81
4	481	15541	4	81
5	481	15541	3	125
6	481	15541	2	100
7	481	15541	4	64
8	481	15541	3	30
9	481	15541	2	48
10	1000	56210	5	3125
11	1000	56210	12	500000
12	2000	58254	15	160000
13	4000	10891	8	6561
14	4000	10891	4	81
15	4000	58254	30	78125
16	8000	58254	40	177147
17	10000	58254	10	337500

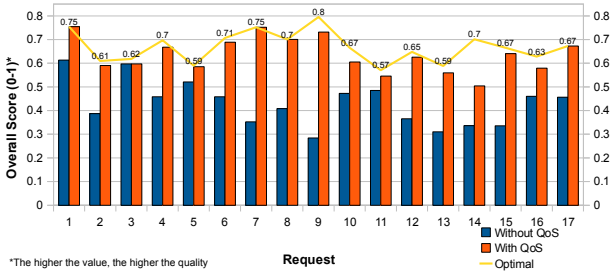

Figure 5: Overall Composition Score.

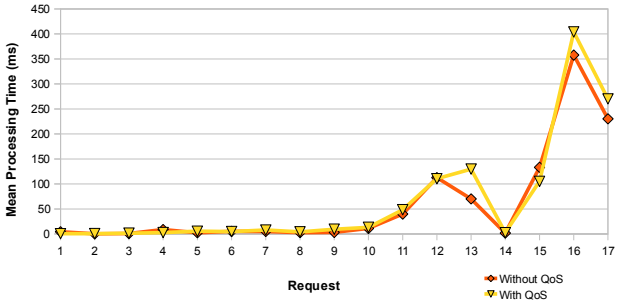

Figure 6: Mean Processing Time.

their Inputs and Outputs parameters with concepts of ontology $\mathcal{O}$ and provide values for each quality criteria, as shown in Table 2.

- The request $\mathcal{R}$ has a set of available inputs $\mathcal{R}_{in} = \{c_5\}$, set of required outputs $\mathcal{R}_{out} = \{c_{10}, c_{19}\}$ and a vector of quality $\mathcal{R}_{qos} = \{(q_{du}, 150, 0.2), (q_{av}, 0.60, 0.2), (q_{rep}, 0.94, 0.2), (q_{pr}, 0.8, 0.2), (q_{rat}, 0.35, 0.20)\}$, where q_{du} is Duration, q_{av} is Availability, q_{rep} is Reputation, q_{pr} is Price and q_{rat} is Successful Execution Rate.

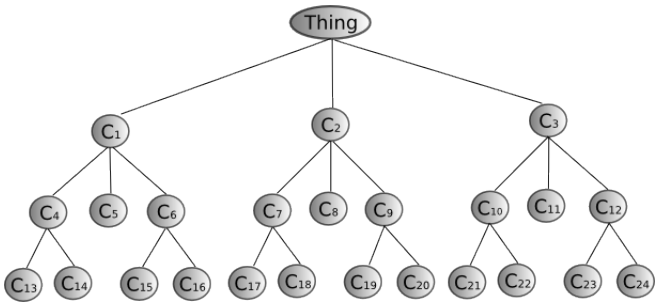

Figure 7: Domain Ontology of the Example Scenario.

head in the processing time, or positive, such as on the improvement of the overall quality of the compositions. The experiments were performed by executing the original algorithm, i. e. the algorithm not considering the QoS criteria, and the algorithm considering QoS criteria under each request of test sets of Table 1. Primarily, several executions for each request were performed in order to measure the mean processing time of both versions of the algorithm. Then, the overall criteria values of each composition found was computed and contrasted in order to determine whether or not a composition is better than other in terms of a specific QoS criterion.

Further details about the experiments can be found in [10].

4.2 Evolving Dynamic Web Service Composition Scenario

This section presents an example scenario of evolving dynamic web service composition in order to detail the proposed approach. The main elements of this scenario are described as follows:

- The domain ontology $\mathcal{O}$, which is given by the taxonomy shown in Figure 7.

- The service repository $\mathcal{D}$, in which services describe

The algorithm starts by trying to find out the services that produce as output concepts that are related under subsumption reasoning to the concepts c_{19} and c_{10}. Service A is selected as a candidate composition, because it provides either c_{19} which is equal to one of the required concepts and c_{21} which is subsumed by the required concept c_{10}. After the first iteration, the list of candidate compositions will have only one candidate composition, as shown in Table 3 where e is the number of eliminated concepts, u is the number of unknown concepts and k is the number of known concepts. Since there is only one candidate composition, this composition will have both the maximum and minimum values for every criteria, and thus will receive the maximum score (1.0) and will be chosen as the most promising composition. As service A provides concepts for both required concepts, the only concept needed at this moment is the concept required to execute the service A, i. e. concept c_{12}. The criteria values of each candidate compositions should be re-calculated in every iteration, since they are based on the maximum and minimum values of the current candidate compositions.

The next iteration will search for service that provide concept c_{12}. Services B, C and D are selected to form three new candidate compositions with service A. So, Table 4 shows the calculated and retrieved values for the candidates compositions $\{B, A\}$, $\{C, A\}$ and $\{D, A\}$ where e is the number of eliminated concepts, u is the number of unknown

Services	s_{in}	s_{out}	q_{du}	q_{av}	q_{rep}	q_{pr}	q_{rat}	**QoS Score**
A	c_{12}	c_{19}, c_{21}	45.00	0.90	0.92	0.15	0.95	-
B	c_{11}	c_{23}	32.00	0.95	0.90	0.10	0.87	-
C	c_7	c_{24}	30.00	0.95	0.90	0.12	0.90	-
D	c_8	c_{12}	35.00	0.85	0.95	0.07	0.90	-
E	c_6	c_8	40.00	0.91	0.97	0.17	0.90	-
F	c_{14}	c_8	30.00	0.89	0.99	0.20	0.95	-
G	c_{13}	c_8	23.00	0.92	0.95	0.40	0.86	-
H	c_5	c_{14}	19.00	0.96	0.86	0.25	0.99	-
I	c_5	c_{14}	16.00	0.98	0.90	0.33	0.94	-
J	c_{15},c_{22}	c_{14}	25.00	0.89	0.94	0.34	0.96	-
B,A	c_{11},c_{12}	c_{19},c_{21},c_{23}	77.00	0.86	0.91	0.25	0.83	0.40
C,A	c_7,c_{12}	c_{19},c_{21},c_{24}	75.00	0.86	0.91	0.27	0.86	0.50
D,A	c_8,c_{12}	c_{19},c_{21},c_{12}	80.00	0.76	0.94	0.22	0.86	0.60

Table 2: Service repository($\mathcal{D}$).

Candidates	e	u	k	q_{du}	q_{av}	q_{rep}	q_{pr}	q_{rat}	**QoS Score**
{A}	2	1	1	45.00	0.90	0.92	0.15	0.95	1.00

Table 3: Candidate compositions in the first iteration.

concepts and k is the number of known concepts. It should be noticed that these candidate compositions are searched in the Service Repository to check their existence and to retrieving their corresponding QoS values. The compositions $\{B, A\}$, $\{C, A\}$ and $\{D, A\}$ already exist and their QoS are retrieved. Again, according to the proposed heuristics (comparator function), the most prospective composition will be determined by the overall QoS Score, since the eliminated concepts e and unknown concepts u are the same for the three candidates. Composition $\{D, A\}$ is then selected. To be completed this composition requires concept c_8.

The next iteration will search for services whose outputs are semantically related to concept c_8. Services E, F and G are selected to form another three compositions along with composition $\{D, A\}$, as shown in Table 5. Now there are two compositions with three eliminated concepts (e) and three compositions with four eliminated concepts and all of them have only one unknown concept (u). Thus, the overall QoS Score is used again as a tiebreaker and composition $\{F, D, A\}$ is selected.

Composition $\{F, D, A\}$ will have only one unknown concept, which, by the way, is the concept required to trigger the execution of service F. The next iteration will then look for services that provide a concept which is semantically related to concept c_{14}. Thus, services H, I and J are selected to form another three compositions with composition $\{F, D, A\}$, as shown in Table 6. It is important to notice that all of these new compositions have the same number of unknown concepts and thus might be valid solutions. The QoS Score is once more used to select composition $\{H; F; D; A\}$ as the most prospective composition. However, this composition does not meet the constraints imposed by the request, since the minimum reputation specified was 0.94, whereas the overall reputation of $\{H; F; D; A\}$ is 0.93 and therefore is not a valid composition. Composition $\{I; F; D; A\}$ should be selected, instead, because it meets the constraints imposed by the request as shown Table 6. Figure 8 represents the final composition. This figure represents services as components with their input and output interfaces. It should be noted that every partial or final compositions are registered in the Service Directory, in order they can be discovered and plugged in other requests, then reducing process time.

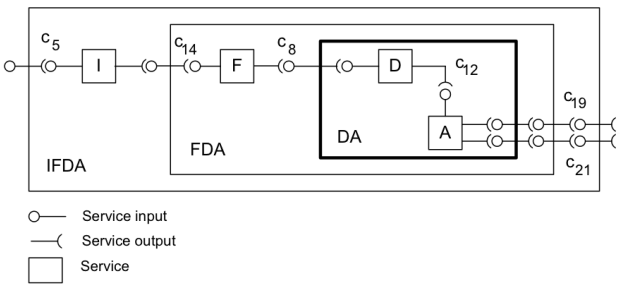

Figure 8: Final Resultant Composition

Candidates	e	u	k	q_{du}	q_{av}	q_{rep}	q_{pr}	q_{rat}	**QoS Score**
{B,A}	3	1	1	77.00	0.86	0.91	0.25	0.83	0.40
{C,A}	3	1	1	75.00	0.86	0.91	0.27	0.86	0.50
{D,A}	3	1	1	80.00	0.76	0.94	0.22	0.86	0.60

Table 4: Candidate compositions in the second iteration.

Candidates	e	u	k	q_{du}	q_{av}	q_{rep}	q_{pr}	q_{rat}	**QoS Score**
{B,A}	3	1	1	77.00	0.86	0.91	0.25	0.83	0.76
{C,A}	3	1	1	75.00	0.86	0.91	0.27	0.86	0.78
{E,D,A}	4	1	1	120.00	0.70	0.95	0.39	0.77	0.39
{F,D,A}	4	1	1	110.00	0.68	0.95	0.42	0.81	0.49
{G,D,A}	4	1	1	103.00	0.70	0.94	0.62	0.74	0.26

Table 5: Candidate compositions in the third iteration.

Candidates	e	u	k	q_{du}	q_{av}	q_{rep}	q_{pr}	q_{rat}	**QoS Score**
{B,A}	3	1	1	77.00	0.86	0.91	0.25	0.83	0.77
{C,A}	3	1	1	75.00	0.86	0.91	0.27	0.86	0.79
{G,D,A}	4	1	1	103.00	0.70	0.94	0.62	0.74	0.43
{E,D,A}	4	1	1	120.00	0.70	0.95	0.39	0.77	0.54
{H,F,D,A}	6	0	2	129.00	0.65	0.93	0.67	0.80	0.26
{I,F,D,A}	6	0	2	126.00	0.67	0.94	0.75	0.76	0.25
{J,F,D,A}	5	2	1	135.00	0.61	0.95	0.76	0.78	0.24

Table 6: Candidate compositions in the forth and last iteration.

5 Conclusion

The semantic web enriches web services functional description by adding ontology-based annotations so that it is possible to computers automatically discover these services and compose them in order to have new functionalities based on existing ones.

In this work, an infrastructure for evolving dynamic web service composition was presented. The infrastructure is divided into two main modules: the execution engine, where the actual services are executed and monitored; and the composer module, where the several semantically annotated services are automatically composed to accomplish a given task. A scenario and experiments were used to show the practical usage and feasibility of the infrastructure.

The use of services' effects and preconditions is left for future works. In addition, other techniques used for search problems like Dynamic Programming [9] should also be studied and compared to the proposed approach.

Questions like parallel composition remain to be investigated. Due to the concurrency, the composition execution time can be significantly reduced.

References

[1] *9th IEEE International Conference on E-Commerce Technology (CEC 2007) / 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2007), 23-26 July 2007, National Center of Sciences*, Tokyo, Japan, 2007. IEEE Computer Society.

[2] R. Akkiraju. Semantic web services. In K. Klinger, K. Roth, J. Neidig, S. Reed, S. Berger, and J. LeBlanc, editors, *Semantic Web Services: Theory, Tools and Applications*, pages 191–216. IGI Global, London, UK/Hershey, PA, USA, 2007.

[3] M. B. Blake, W. K. Cheung, M. C. Jaeger, and A. Wombacher. Wsc-07: Evolving the web services challenge. *E-Commerce Technology, IEEE International Conference on, and Enterprise Computing, E-Commerce, and E-Services, IEEE International Conference on*, 0:505–508, 2007.

[4] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*, April 2000.

[5] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. Qos-aware replanning of composite web services. *Web Services, IEEE International Conference on*, 0:121–129, 2005.

[6] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava. Adaptation inweb service composition and execution. *Web Services, IEEE International Conference on*, 0:549–557, 2006.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. W3C note, W3C, Mar. 2007.

[8] D. B. Claro, O. Licchelli, P. Albers, and R. J. de Araújo Macêdo. Personalized reliable web service compositions. In F. L. G. de Freitas, H. Stuckenschmidt, H. S. Pinto, A. Malucelli, and Ó. Corcho, editors, *WONTO*, volume 427 of *CEUR Workshop Proceedings*, Salvador-Bahia, Brazil, 2008. CEUR-WS.org.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, USA, 1990.

[10] F. G. A. de Oliveira Jr. A qos-based approach for dynamic web service composition. Master's thesis, Instituto Tecnológico de Aeronáutica, São José dos Campos, Brasil, 2009.

[11] F. G. A. de Oliveira Jr. and J. M. P. de Oliveira. A heuristic-based runtime ranking for service composition. In *ICITST-2009: International Conference for Internet Technology and Secured Transactions*, London, UK, 2009. IEEE Computer Society.

[12] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[13] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive qos monitoring of web services and event-based sla violation detection. In *MWSOC '09: Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, pages 1–6, New York, NY, USA, 2009. ACM.

[14] OASIS. Universal description, discovery, and integration (UDDI) version 3.0.2, April 2004.

[15] S.-C. Oh, J.-W. Yoo, H. Kil, D. Lee, and S. R. T. Kumara. Semantic web-service discovery and composition using flexible parameter matching. *E-Commerce Technology, IEEE International Conference on, and Enterprise Computing, E-Commerce, and E-Services, IEEE International Conference on*, 0:533–542, 2007.

[16] J. O'Sullivan, D. Edmond, and A. T. Hofstede. What's in a service? *Distrib. Parallel Databases*, 12(2-3):117–133, 2002.

[17] T. Weise, S. Bleul, D. Comes, and K. Geihs. Different approaches to semantic web service composition. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 90–96, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Y. Yan, B. Xu, and Z. Gu. Automatic service composition using and/or graph. *E-Commerce Technology and Enterprise Computing, E-Commerce and E-Services, IEEE Conference and Fifth IEEE Conference*, 0:335–338, 2008.

[19] P. K. Yoon, C.-L. Hwang, and K. Yoon. *Multiple Attribute Decision Making: An Introduction (Quantitative Applications in the Social Sciences)*. Sage Publications Inc, Thousand Oaks, CA, USA, March 1995.

[20] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.