Hybrid Optimization Algorithm for Large-Scale QoS-Aware Service Composition

Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama

Abstract—In this paper we present a hybrid approach for automatic composition of Web services that generates semantic input-output based compositions with optimal end-to-end QoS, minimizing the number of services of the resulting composition. The proposed approach has four main steps: 1) generation of the composition graph for a request; 2) computation of the optimal composition that minimizes a single objective QoS function; 3) multi-step optimizations to reduce the search space by identifying equivalent and dominated services; and 4) hybrid local-global search to extract the optimal QoS with the minimum number of services. An extensive validation with the datasets of the Web Service Challenge 2009-2010 and randomly generated datasets shows that: 1) the combination of local and global optimization is a general and powerful technique to extract optimal compositions in diverse scenarios; and 2) the hybrid strategy performs better than the state-of-the-art, obtaining solutions with less services and optimal QoS.

Keywords—Service Composition; Service Optimization; Hybrid Algorithm; QoS-aware; Semantic Web Services.

1 Introduction

Web services are self-describing software applications that can be published, discovered and invoked accross the Web using standard technologies [1]. The functionality of a Web service is mainly determined by the functional properties that describe their behaviour in terms of its inputs, outputs, and also possibly additional descriptions that the services may have, such as preconditions and effects. These four characteristics, commonly abbreviated IOPEs, allow the composition and aggregation of Web services into composite Web services that achieve more complex functionalities and, therefore, solve complex user needs that cannot be satisfied with atomic Web services. However, compositions should go beyond achieving a concrete functionality and take into account other requirements such as Quality-of-Service (QoS) to generate also compositions that fit the needs of different contexts. The QoS determines the value of different quality properties of services such as response time (total time a service takes to respond to a request) or throughput (number of invocations supported in a given time interval), among others characteristics. These properties apply both to single services and to composite services, where each individual service in the composition contributes to the global QoS. For composite services this implies that having many different services with similar or identical functionality, but different QoS, may lead to a large amount of possible compositions that satisfy the same functionality with different QoS but also with a different number of services.

However, the problem of generating automatic compositions that satisfy a given request with an optimal QoS is a very complex task, specially in large-scale environments, where many service providers offer services with similar functionality but with different QoS. This has motivated researchers to explore efficient strategies to generate QoS-aware Web service compositions from different perspectives [2], [3]. But despite the large number of strategies proposed so far, the problem of finding automatic compositions that minimize the number of services while guaranteeing the optimal end-to-end QoS is rarely considered. Instead, most of the work has focused on optimizing the global QoS of a composition or improving the execution time of the composition engines. An analysis of the literature shows that only a few works take into consideration the number of services of the resulting optimal QoS compositions. Some notable examples are [4]–[7]. Although most of these composition engines are quite efficient in terms of computation time, none of them are able to effectively minimize the total number of services of the solution while keeping the optimal QoS.

The ability to provide not only optimal QoS but also an optimal number of services is specially important in large-scale scenarios, where the large number of services and the possible interactions among them may lead to a vast amount of possible solutions with different number of services but also with the same optimal QoS for a given problem. Moreover, there can be situations where certain QoS values are missing or cannot be measured. Although the prediction of QoS can partially alleviate this problem [8], it is not always possible to have historical data in order to build statistical models to accurately predict missing QoS. In this context, opti-

- *P. Rodriguez-Mier, M. Mucientes and M. Lama work at the Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Spain.*
E-mail: {pablo.rodriguez.mier,manuel.mucientes,manuel.lama}@usc.es

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

mizing not only the available QoS but also the number of services of the composition may indirectly improve other missing properties. This has important benefits for brokers, customers and service providers. From the broker point of view, the generation of smaller compositions is interesting to achieve manageable compositions that are easier to execute, monitor, debug, deploy and scale. On the other hand, customers can also benefit from smaller compositions, specially when there are multiple solutions with the same optimal end-to-end QoS but different number of services. This is even more important when service providers do not offer fine-grained QoS metrics, since decreasing the number of services involved in the composition may indirectly improve other quality parameters such as communication overhead, risk of failure, connection latency, etc. This is also interesting from the perspective of service providers. For example, if the customer wants the cheapest composition, the solution with fewer services from the same provider may also require less resources for the same task.

However, one of the main difficulties when looking for optimal solutions is that it usually requires to explore the complete search space among all possible combinations of services, which is a hard combinatorial problem. In fact, finding the optimal composition with the minimum number of services is NP-Hard (see Appendix A). Thus, achieving a reasonable trade-off between solution quality and execution time in large-scale environments is far from trivial, and hardly achievable without adequate optimizations.

In this paper we focus on the automatic generation of semantic input-output compositions, minimizing both a single QoS criterion and the total number of services subject to the optimal QoS. The main contributions are:

- A multi-step optimization pipeline based on the analysis of non-relevant, equivalent and dominated services in terms of interface functionality and QoS.
- A fast local search strategy that guarantees to obtain a near-optimal number of services while satisfying the optimal end-to-end QoS for an input-output based composition request.
- An optimal combinatorial search that can improve the solution obtained with the local search strategy by performing an exhaustive combinatorial search to select the composition with the minimum number of services for the optimal QoS.

We tested our proposal using the Web Service Challenge 2009-2010 datasets and, also, a different randomly generated dataset with a variable number of services. The rest of the paper is organized as follows: Sec. 4 introduces the composition problem, Sec. 5 describes the proposed approach, Sec. 6 presents the results obtained, and Sec. 7 gives some final remarks.

2 Related Work

Automatic composition of services is a fundamental and complex problem in the field of Service Oriented Computing, which has been approached from many different perspectives depending on what kinds of assumptions are made [2], [3], [9], [10]. AI Planning techniques have been traditionally used in service composition to generate valid composition plans by mapping services to actions in the planning domain [11]–[16]. These techniques work under the assumption that services are complex operators that are well defined in terms of IOPEs, so the problem can be translated to a planning problem and solved using classical planning algorithms. Most of these approaches have been mainly focused on exploiting semantic techniques [13], [16], [17] and developing heuristics [15], [16], [18] to improve the performance of the planners. As a result, and partly given by the complexity of generating satisfiable plans in the planning domain, these approaches do not generate neither optimal plans (minimizing the number of actions) nor optimal QoS-aware compositions.

Other approaches have studied the QoS-aware composition problem from the perspective of Operation Research, providing interesting strategies for optimal selection of services and optimizing the global QoS of the composition subject to multiple QoS constraints. A common strategy is to reduce the composition problem to a combinatorial Knapsack-based problem, which is generally solved using constraint satisfaction algorithms (such as Integer Programming) [19]–[23] or Evolutionary Algorithms [24], [25]. Some relevant approaches are [19], [22]. In [19] the authors present AgFlow, a QoS middleware for service composition. They analyze two different methods for QoS optimization, a local selection and a global selection strategy. The second strategy is able to optimize the global end-to-end QoS of the composition using a Integer Linear Programming method, which performs better than the suboptimal local selection strategy. Similarly, in [22] the authors propose a hybrid QoS selection approach that combines a global optimization strategy with local selection for large-scale QoS composition. The assumption made by all these approaches is that there is only one composition workflow with a fixed set of abstract tasks, where each abstract task can be implemented by a concrete service. Both the composition workflow and the service candidates for each abstract task are assumed to be prefined beforehand, so these techniques are not able to produce compositions with variable size.

A different category of techniques are graph-based approaches that 1) generate the entire composition by selecting and combining relevant services and 2) optimize the global QoS of the composition. These techniques usually combine variants or new ideas inspired by different fields, such as AI Planning, Operations Research or Heuristic Search, in order to resolve more efficiently the automatic QoS composition, usually for a single QoS criterion. Some relevant approaches in this category are the top-3 winners of the Web Service Challenge (WSC) 2009-2010 [4]–[6]. Concretely, the winners of the WSC challenge [4], presented an approach that automatically

discovers and composes services, optimizing the global QoS. This approach also includes an optimization phase to reduce the number of services of the solution. Although the proposed algorithm has in general good performance, as demonstrated in the WSC, it cannot guarantee to obtain optimal solutions in terms of number of services. The other participants of the WSC have also the same limitation.

A recent and interesting approach in this category has been recently presented by Jiang et al. [26]. In this paper, the authors analyze the problem of generating top K query compositions by relaxing the optimality of the QoS in order to introduce service variability. However, the compositions are generated at the expense of worsening the optimal QoS, instead of looking first for all possible composition alternatives with the minimum number of services that guarantee the optimal QoS.

Another interesting graph-based approach has been presented in [7]. In this paper, the authors propose a service removal strategy that detects services that are redundant in terms of functionality and QoS. Results show that service removal techniques can be very effective to reduce the number of services before extracting the final composition, as anticipated by other similar approaches [27]–[29]. However, some important limitations of this work are: 1) The QoS is not always optimal, since the graph generated for the composition is not complete as it does not contain all the relations between services (it is acyclic) and 2) although the redundancy removal is an effective technique that can be used also to prune the search space, this strategy itself cannot provide optimal results in terms of number of services, and it should be combined with exhaustive search to improve the solutions obtained.

In summary, despite the large number of approaches for automatic QoS-aware service composition there is a lack of efficient techniques that are not only able to optimize the global end-to-end QoS, but also effectively minimize the number of services of the composition. This paper aims to provide an efficient graph-based approach that uses a hybrid local-global optimization algorithm in order to find optimal compositions both in terms of single QoS criteria and in terms of minimum number of services.

3 Motivation

The aim of the automatic service composition problem, as considered in this paper, is to automatically select the best combination of available QoS-aware services in a way that can fulfil a user request that otherwise could not be solved by just invoking a single, existing service. This request is specified in terms of the information that the user provides (inputs), and the information it expects to obtain (outputs). The resulting composition should meet this request with an optimal, single criterion end-to-end QoS and using as less services as possible.

A motivating example of the problem is shown in Fig. 1. The figure represents a graph with all the relevant services for a request R where the inputs are $\{$*ont3:IPAddress*, *ont2:MerchantCode*$\}$ and the output is $\{$*xsd:boolean*$\}$. The goal of this example is to obtain a composition to predict whether a business transaction is fraudulent or not. Each service (associated to a response time QoS) is represented by squares. Inputs and outputs are represented by circles. The graph also contains edges connecting outputs and inputs. These edges represent valid semantic *matches* whenever an output of a service can be passed as an input of a different service. As can be seen, there are some inputs (*ont1:Location*,*ont3:Payment*) that can be matched by more than one output, so there are many different ways to combine services to achieve the same goal.

Although finding the proper combination of services in terms of their inputs/outputs is essential to generate a solution, it is not enough to obtain good compositions, since there can exist different combinations of services with different QoS. Moreover, many different combinations of services may produce compositions with a different number of services but the same end-to-end QoS. For example, in Fig. 1 we can select *WS E-Payment* service or the *Secure Payment* service to process the electronic payment. However, the second service has a higher response time. Using this leads to a sub-optimal end-to-end QoS of 420 ms. However, there are other situations where the selection of different services leads to compositions with different size but same end-to-end QoS. For example, both *Free Geoloc Service* or the *Premium Geoloc Service* can be selected to translate an *IP* to a *Location*. Although the second one has a better average response time (40 ms), it requires an additional service to obtain the *ClientID* for verification purposes. However, selecting the *Premium Geoloc Service* or the *Free Geoloc Service* does not have an impact on the global QoS, since the *ML Predictor Service* has to wait longer to obtain the *Transaction* parameter (200 ms), but it has an impact on the total number of services of the solution.

The goal of this paper is to automatically generate, given a composition request, a graph like the one represented in Fig. 1 as well as to extract the optimal end-to-end QoS composition with the minimum number of services from that graph.

4 Problem Formulation

We herein formalize the main concepts and assumptions regarding the composition model used in our approach, which consists of a semantic, graph-centric representation of the service composition. These concepts are captured in three main models: 1) a service model, which is used to represent services and define how services can be connected or matched to generate composite services; 2) a graph-based composition model, which is used to represent both service interactions and compositions; and 3) a QoS computation model, which provides the

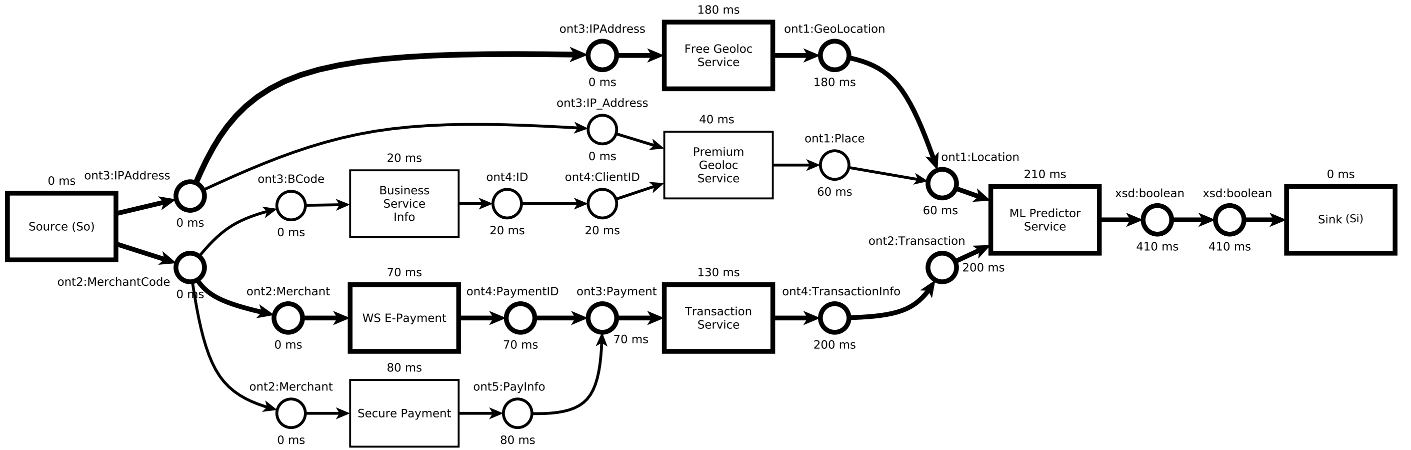

Fig. 1. Example of a *Service Match Graph* for a request $R = \{\{$*ont3:IPAddress*, *ont2:MerchantCode*$\}, \{$*xsd:boolean*$\}\}$ to predict whether a business transaction is fraudulent or not. Each service is associated with an average response time. The optimal solution (*Service Composition Graph*), with an overall response time of 410 ms and 4 services (excluding So and Si) is highlighted.

operators required to compute the global QoS in a graph-based composition.

4.1 Semantic Service Model

The automatic composition of services requires a mechanism to select appropiated services based on their functional descriptions, as well as to automatic match the services together by linking their inputs and outputs to generate executable data-flow compositions. To this end, we introduce here the main concepts that we use in this paper to support the automatic generation of compositions. This model is an extension of a previous model used in [30] to include QoS properties.

Definition 1. *A Composition Request R is defined as a tuple $R = \{I_R, O_R\}$, where I_R is the set of provided inputs, and O_R the set of expected outputs. Each input and output is related to a semantic concept from the set C of the concepts defined in an ontology Ont ($In_w, Out_w \subseteq C$). We say that a composition satisfies the request R if it can be invoked with the inputs in I_R and returns the outputs in O_R.*

Definition 2. *A Semantic Web Service (hereafter "service") can be defined as a tuple $w = \{In_w, Out_w, Q_w\} \in W$ where In_w is a set of inputs required to invoke w, Out_w is the set of outputs returned by w after its execution, $Q_w = \{q_w^1, \dots, q_w^n\}$ is the set of QoS values associated to the service, and W is the set of all services available in the service registry.*

Each input and output is related to a semantic concept from the set C of the concepts defined in an ontology Ont ($In_w, Out_w \subseteq C$). Each QoS value $q_w^i \in Q_w$ has a concrete type associated to a set of valid values Q. For example, the QoS values of a service w with two different measures, an average response time of 20 ms and an average throughput of 1000 invocations/second, is represented as $Q_w = \{20ms, 1000\ inv/s\}$, where $20ms \in Q_{RT}$ and $1000\ inv/s \in Q_{TH}$.

Semantic inputs and outputs are used to compose the functionality of multiple services by matching their inputs and outputs together. In order to measure the quality of the match, we need a matchmaking mechanism that exploits the semantic I/O information of the services. The different matchmaking degrees that are contemplated are *exact*, *plugin*, *subsumes* and *fail* [31].

Definition 3. *Given $a, b \in C$, degree(a,b) returns the degree of match between both concepts (exact, plugin, subsume or fail), which is determined by the logical relationship of both concepts within the Ontology.*

Definition 4. *Given $a, b \in C$, match(a,b) holds if $degree(a, b) \neq fail$.*

In order to determine which concepts are matched by other concepts, we define a matchmaking operator "$\otimes$" that given two sets of concepts $C_1, C_2 \subseteq C$, it returns the concepts from C_2 matched by C_1.

Definition 5. *Given $C_1, C_2 \subseteq C$, we define "$\otimes : C \times C \rightarrow C$" such that $C_1 \otimes C_2 = \{c_2 \in C_2 | match(c_1, c_2), c_1 \in C_1\}$.*

We can use the previous operator to define the concepts of full and partial matching between concepts.

Definition 6. *Given $C_1, C_2 \subseteq C$, a full matching between C_1 and C_2 exists if $C_1 \otimes C_2 = C_2$, whereas a partial matching exists if $C_1 \otimes C_2 \subset C_2$.*

Definition 7. *Given a set of concepts $C' \subseteq C$, a service $w = \{In_w, Out_w\}$ is invokable if $C' \otimes In_w = In_w$, i.e., there is a full match between the provided set of concepts C' and In_w, so the information required by w is fully satisfied.*

This internal model used by the algorithm, which captures the core components required to perform semantic matchmaking and composition of services, is agnostic to how semantic services are represented. Thus, the algo-

rithm is not bound to any concrete service description. Concretely, different service descriptions can be handled by the algorithm through the use of iServe importers for OWL-S, WSMO-lite, SAWSDL or MicroWSMO. For further details see [32].

4.2 Graph-Based Composition Model

In a nutshell, a data-flow composition of services can be seen as a set of services connected together through their inputs and output, using the semantic model defined before, in a way that every service in the composition is invocable and the invocation of each service in the composition can *transform* a set of inputs into a set of outputs. These concepts can be naturally captured by graphs, where the vertices represent inputs, outputs and services, and the edges represent semantic matches between inputs and outputs. Here we define the notion of *Service Match Graph* and *Service Composition Graph*. The *Service Match Graph* is a graph that captures all the existent dependencies (matches) between all the relevant services for a composition request. The *Service Composition Graph* is a particular case of the *Service Match Graph* that represents a composition contained in the *Service Match Graph*.

The *Service Match Graph* represents the space of all possible valid solutions for a composition request R, and it is defined as a directed graph $G_S = (V, E)$, where:

- $V = W_R \cup I \cup O \cup \{So, Si\}$ is the set of vertices of the graph, where $W_R \subseteq W$ is the set of relevant services, I is the set of inputs and O is the set of outputs. Si and So are two special services, called *Source* and *Sink* defined as $So = \{\emptyset, I_R\}$, $Si = \{O_R, \emptyset\}$.
- $E = IW \cup WO \cup OI$ is the set of edges in the graph where:
  - $IW \subseteq \{(i_w, w) \mid i_w \in I \wedge w \in W\}$ is the set of input edges, i.e., edges connecting input concepts to their services.
  - $WO \subseteq \{(w, o_w) \mid w \in W \wedge o_w \in O\}$ is the set of output edges, i.e., edges connecting services with their output concepts.
  - $OI \subseteq \{(o_w, i_{w'}) \mid o_w, i_{w'} \in (I \cup O) \wedge match(o_w, i_{w'})\}$ is the set of edges that represent a semantic match between an output of w and an input of w'.

There are also some restrictions in the edge set to ensure that each input/output belongs to a single service:

- $\forall i \in I\ d^+_{G_S}(i) = 1 \wedge ch_{G_S}(i) = \{w\}, w \in W$ (each input has only one outgoing edge which connects the input with its service)
- $\forall o \in O, d^-_{G_S}(o) = 1 \wedge par_{G_S}(o) = \{w\}, w \in W$ (each output has only one incoming edge which connects the output with its service)

Function $d^+_{G_S}(v)$ returns the outdegree of a vertex $v \in G_S$ (number of children vertices connected to v), whereas $d^-_{G_S}(v)$ returns the indegree of a vertex v (number of parent vertices connected to v). The functions $ch_G(v)$ and $par_G(v)$ are the functions that returns the children vertices of v and the parent vertices of $v \in G_S$, respectively.

Fig. 1 shows an example of a *Service Match Graph* where each service is associated with its average response time. As can be seen, this graph contains many different compositions since there are inputs in the graph that can be matched by the outputs of different services. For example, the parent nodes of the input *ont1:Location* of the service *ML Service Predictor* (par_G(*ont1:Location*)) in Fig. 1 are *ont1:GeoLocation* and *ont1:Place*, so the input is matched by two outputs $d^-_{G_S}(\textit{ont1:Location}) = 2$.

A *Service Composition Graph*, denoted as $G_C = (V, E)$, represents a solution for the composition request where each input is exactly matched by one output. Formally, it is a subgraph of *Service Match Graph* ($G_C \subseteq G_S$) that satisfies the following conditions:

- $\forall i \in I, d^-_{G_C}(i) = 1$ (each input is strictly matched by one output)
- G_C is a Directed Acyclic Graph (DAG)

These conditions are important in order to guarantee that a solution is valid, i.e, each input is matched by an output of a service and each service is invocable (all inputs on the composition are matched with no cyclic dependencies). This definition of service composition is language-agnostic, so the resulting DAG is a representation of a solution for the composition problem which can be translated to a concrete language, such as OWL-S or BPEL.

4.3 QoS Computation Model

Before looking for optimal QoS service compositions, we need first to define a model to work with QoS over compositions of services which allow us to determine the best QoS that can be achieved for a given composition request on a service repository. When many services are chained together in a composition, the QoS of each individual service contributes to the global QoS of the composition. For example, suppose we want to measure the total response time of a simple composition with two services chained in sequence. The total response time is calculated as the sum of the response time of each service in the composition. However, if the composition has two services in parallel, the total time of the composition is given by the slowest services. Thus, the calculation of the QoS of a composition depends on the type of the QoS and on the structure of the composition.

In order to define the common rules to operate with QoS values in composite services, many approaches use a QoS computation model based on workflow patterns [33], which is adequate to measure the QoS of control-flow based compositions. However, this paper focuses on the automatic generation of optimal QoS-aware compositions driven by the data-flow analysis of the service dependencies (input-output matches) that are represented as a *Service Match Graph*.

In this section we explain the general graph-centric QoS computation model that we use, based on the *path algebra* defined in [34]. This model is better suited to compute QoS values in a *Service Match Graph*, which, for extension, is also applicable to the particular case of the *Service Composition Graph*.

Definition 8. $(Q, \oplus, \ominus, \preceq)$ *is a QoS algebraic structure to operate with a set of QoS values, denoted as* Q*. This set is equipped with the following elements:*

- $\oplus : Q \times Q \rightarrow Q$ *is a closed binary operation for aggregating QoS values*
- $\ominus : Q \times Q \rightarrow Q$ *is a binary operation for subtracting QoS values*
- $\preceq$ *is a total order relation on* Q

This algebraic structure has the following properties:

1) Q is closed under $\oplus$ (any aggregation of two QoS values always returns a QoS value)
2) The set Q contains an identity element e such that $\forall a \in Q, a \oplus e = e \oplus a = a$
3) The set Q contains a zero element ϕ such that $\forall a \in Q, \phi \oplus a = a \oplus \phi = \phi$
4) The operator $\oplus$ is associative
5) The operator $\oplus$ is monotone for $\preceq$ (preserves order). This implies that $\forall a, b, c \in Q, a \preceq b \Leftrightarrow a \oplus c \preceq b \oplus c$
6) The operator $\ominus$ is the inverse of $\oplus$: $a \ominus b = c \Leftrightarrow a = c \oplus b$

Table 1 shows an example of the concrete elements in this algebra. Note that, for the sake of brevity, only the response time and throughput operators are represented in Table 1. However, other QoS properties such as cost, availability, reputation, etc, can also be defined by instantiating the corresponding operators. We denote Q_{RT} the set of QoS values for response time (in milliseconds), Q_{TH} the set of QoS values for throughput (invocations/second). The total order comparator $\preceq$ is required to be able to order and compare different QoS values. Given two QoS values $a, b \in Q$, $a \preceq b$ means that a is equal or ***better*** than b, whereas $b \preceq a$ means that a is equal or ***worse*** than b. The order depends on the concrete comparator defined on Q. For example, Q_{RT} uses the comparator $\leq$ to order the response time, so $a, b \in Q_{RT}$, $a \preceq b \Leftrightarrow a \leq b$. For example, given two response times $10ms, 20ms \in Q_{RT}$, $10ms \prec 20ms$ ($10ms$ is better than $20ms$) since $10ms < 20ms$. However, Q_{TH} uses the comparator $\geq$, so $a, b \in Q_{TH}$, $a \preceq b \Leftrightarrow a \geq b$. For example, given two throughput values 10 *inv/s*, 20 *inv/s* $\in Q_{TH}$, 20 *inv/s* $\prec$ 10 *inv/s* (20 *inv/s* is better than 10 *inv/s*) since 20 *inv/s* $>$ 10 *inv/s*. This order relation also affects the behavior of the min and max functions. The min function always selects the *best* QoS value, whereas the max function always selects the *worst* QoS value.

Definition 9. $F_Q(w) : W \rightarrow Q$ *is a function that given a service* $w \in W$*, it returns its corresponding QoS value from* Q_w *with type* Q*. This function can be seen as a function to measure the QoS of a service.*

TABLE 1
QoS algebra elements for response time and throughput

QoS (Q)	$a \oplus b$	$a \ominus b$	e	ϕ	Order ($\preceq$)
$Q_{RT} = \mathbb{R}_{\geq 0} \cup \{\infty\}$	a + b	a - b	0	∞	$\leq$
$Q_{TH} = \mathbb{R}_{\geq 0} \cup \{\infty\}$	min(a, b)	min(a,b)	∞	0	$\geq$

For example, in Fig. 1, $F_{Q_{RT}}(\textit{Trans. Service}) = 130ms$.

Definition 10. $V_Q(w) : W \rightarrow Q$ *is a function that given a service* w*, it returns its aggregated QoS value. This is defined as:*

$$V_Q(w) = \begin{cases} \max\limits_{\forall i \in In_w} (V_Q^{in}(i)) \oplus F_Q(w) & \text{if } In_w \neq \emptyset \\ F_Q(w) & \text{if } In_w = \emptyset \end{cases} \quad (1)$$

Informally, this function calculates the aggregated QoS of a service by taking the worst value of the QoS of its inputs plus the current QoS value of the service itself. Taking for example the service *Premium Geoloc Service* from Fig. 1, $V_{Q_{RT}}$(*Premium Geoloc Service*) is computed as $max(V_{Q_{RT}}^{in}(\textit{ont3:IP_Address}), V_{Q_{RT}}^{in}(\textit{ont4:ClientID})) \oplus 40ms$, which is $max(0ms, 20ms) \oplus 40ms = 60ms$ (see Def. 12).

Definition 11. $V_Q^{out}(o_w) : O \rightarrow Q$ *is a function that given an output of a service* w*,* $o_w \in O$*, it returns its aggregated QoS value. The aggregated QoS of an output is equal to the aggregated QoS of a service. Thus, it is defined as:*

$$V_Q^{out}(o_w) = V_Q(w) \quad (2)$$

For example, the aggregated QoS of the output *ont1:Place* ($V_{Q_{RT}}^{out}(\textit{ont1:Place})$) is equal to the aggregated QoS of its service *Premium Geoloc Service* ($V_{Q_{RT}}$(*Premium Geoloc Service*)), which is equal to *60ms*.

Definition 12. $V_Q^{in}(i_w) : I \rightarrow Q$ *is a function that given an input of a service* w*,* $i_w \in I$*, it returns its optimal aggregated QoS value. This function is defined as:*

$$V_Q^{in}(i_w) = \begin{cases} \phi & \text{if } d_{G_S}^{-}(i_w) = 0 \\ V_Q^{out}(o_{w'}), o_{w'} \in par_G(i_w) & \text{if } d_{G_S}^{-}(i_w) = 1 \\ \min\limits_{\forall o_{w'} \in par_G(i_w)} (V_Q^{out}(o_{w'})) & \text{if } d_{G_S}^{-}(i_w) > 1 \end{cases} \quad (3)$$

Given an input $i_w \in In_w$ of a service w, this function returns the accumulated QoS for that input. If the evaluated input is not matched by any output ($d_{G_S}^{-}(i_w) = 0$), then the accumulated QoS of the input is undefined. If the evaluated input is matched by just one output ($d_{G_S}^{-}(i_w) = 1$), then its accumulated QoS value is equal to the accumulated QoS of that output. If the evaluated input can be matched by more than one output ($d_{G_S}^{-}(i_w) > 1$), i.e., there are many services that can match that input, then its accumulated QoS value is computed by selecting the optimal (best) QoS.

For example, the optimal aggregated QoS of the input *ont3:Payment* from *Transaction*

Service ($V^{in}_{Q_{RT}}$(*ont3:Payment*)) is calculated as $\min(V^{out}_{Q_{RT}}(\textit{ont3:PaymentID}), V^{out}_{Q_{RT}}(\textit{ont5:PayInfo})) = 70ms$.

Definition 13. *We define $V^G_Q(g) : G \to Q$ as a function that given a Service Match Graph $g = (V, E)$, it returns its optimal aggregated QoS value. This is defined as:*

$$V^G_Q(g) = V_Q(Si), S_i \in V \tag{4}$$

Basically, the optimal QoS of a *Service Match Graph* G_S corresponds with the optimal aggregated QoS of its service $Si \in G_S$.

4.4 Composition Problem

Given a composition request $R = \{I_R, O_R\}$, a set of semantic services W, a semantic model and a QoS algebra, the composition problem considered in this paper consists of generating the *Service Match Graph* G_S and selecting a composition graph $G_C \subset G_S$ such that:

1) $\forall G'_C, V^G_Q(G_C) \leq V^G_Q(G'_C)$, i.e., the composition graph has the best possible QoS
2) $W_R \subseteq V, |W_R|$ is minimized (the composition graph contains the minimum number of services)

5 Composition Algorithm

On the basis of the formal definition of the automatic QoS-aware composition problem, in this section we present our hybrid approach strategy for automatic, large-scale composition of services with optimal QoS, minimizing the services involved in the composition. The approach works as follows: given a request, a directed graph with the relevant services for the request is generated. Once the graph is built, an optimal label-correcting forward search is performed in polynomial time in order to compute the global optimal QoS. This information is used later in a multi-step pruning phase to remove suboptimal services. Finally, a hybrid local/global search is performed within a fixed time limit to extract the optimal solution from the graph. The local search returns a near-optimal solution fast whereas the global search performs an incremental search to extract the composition with the minimum number of services in the remaining time. In this section we explain each step of the algorithm, namely: 1) generation of the *Service Match Graph*; 2) calculation of the optimal end-to-end QoS; 3) multi-step graph optimizations and 4) hybrid algorithm.

5.1 Generation of the Service Match Graph

Given a composition request, which specifies the inputs provided by the user as well as the outputs it expects to obtain, and a set of available services, the first step consists of locating all the relevant services that can be part of the final composition, as well as computing all possible matches between their inputs and outputs, according to the semantic model presented in Sec. 4.1. The output of this step is a *Service Match Graph* that

```
1:  function SERVICEMATCHGRAPH(R = {I_R, O_R}, W)
2:      C := I_R; W' := W; W_R := {So, Si}
3:      unmatchedIn := [ ]; availCon := I_R
4:      repeat
5:          W_selected = ∅
6:          W_rel := {w ∈ W' | availCon ⊗ In_w ≠ ∅}
7:          W_rel := W_rel \ W_R
8:          for all w_i = {In_wi, Out_wi} ∈ W_rel do
9:              U_set := unmatchedIn[w_i]
10:             M_set := C ⊗ U_set
11:             unmatchedIn[w_i] := U_set \ M_set
12:             if M_set = ∅ then
13:                 W_selected = W_selected ∪ w_i
14:                 availCon := availCon ∪ Out_wi
15:         W' := W' \ W_selected
16:         W_R := W_R ∪ W_selected
17:         C := C ∪ availCon
18:         availCon := ∅
19:     until W_selected = ∅
20:     return COMPUTE-GRAPH(W_R)
```

Fig. 2. Algorithm for generatig a Service Match Graph from a composition request R and a set of services W.

contains many possible valid compositions for the request, as the one represented in Fig. 3. In a nutshell, the generation of the graph is calculated by selecting all invocable services layer by layer, starting with So in the first layer (the source service whose outputs are the inputs of the request) and terminating with Si in the last layer (the sink service whose inputs are the outputs of the request) [35].

The pseudocode of the algorithm is shown in Fig. 2. The algorithm runs in polynomial time, selecting $W_{selected} \subseteq W$ services at each step. At each layer, the algorithm finds a potential set of relevant services whose inputs are matched by some outputs generated in the previous layer using the $\otimes$ operator (L.6). Then, for each potential eligible service, the algorithm checks whether the service is invokable or not (i.e., all its inputs are matched by outputs of previous layers) by checking if all the unmatched inputs of the service are matches. All the inputs that are matched are removed from the unmatched set of inputs for the current service (L.11). If the service is invokable (has no unmatched inputs), it is selected and its outputs are added to the set of the available concepts. In case the service still has some unmatched inputs, these inputs are stored in a map to check it again in the next layer. For example, the first eligible services for the request shown in Fig. 3 are the services in the layer $L1$, which correspond with the services whose inputs are fully matched by I_R (the set of output concepts produced in $L0$). The second eligible services are those services (placed in $L2$) whose inputs are fully matched by the outputs of the previous layers, and so on. The algorithm stops when no more services are added to the set of selected services. Finally, *COMPUTE-GRAPH*

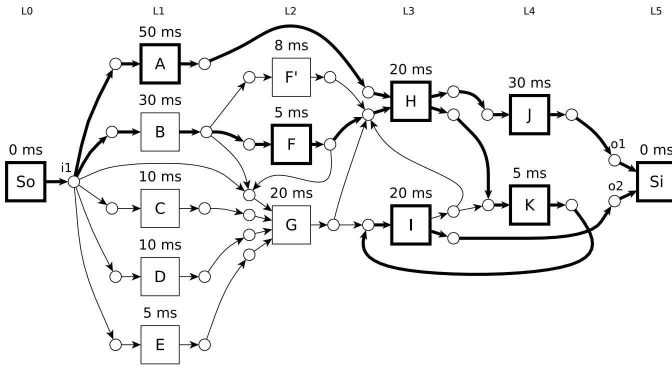

Fig. 3. Graph example with the solution with optimal QoS and minimum number of services highlighted.

computes all possible matches between the outputs and the inputs of the selected services. The output of this process is a complete *Service Match Graph* that can contain cycles, as the one depicted in Fig. 3.

5.2 Optimal end-to-end QoS

Once the *Service Match Graph* is computed for a composition request, the next step is to calculate the best end-to-end QoS achievable in the *Service Match Graph*. The optimal end-to-end QoS can be computed in polynomial time using a shortest path algorithm to calculate the best aggregated QoS values for each input and output of the graph, i.e., the best QoS values at which the outputs can be generated and the inputs are matched. In order to compute the optimal QoS, we use a generalized Dijkstra-based label-setting algorithm computed forwards from So to Si [36], based on the algebraic model of the QoS presented in Sec. 4. The optimality of the algorithm is guaranteed as long as the function defined to aggregate the QoS values ($\otimes$) is monotonic, in order to satisfy the principle of optimality. A proof can be found in [37].

Fig. 4 shows the pseudocode of the generalized Dijkstra-based label-setting algorithm. The algorithm starts assigning infinite QoS cost to each input in the graph in the table qos. An infinite cost for an input means that the input is still not *resolved*. The first service to be processed is So. Each time a service w is processed from the queue, the best accumulated QoS cost of each input $i_{w'}$ matched by the outputs of the service w is recalculated. If there is an improvement (i.e., a match with a better QoS is discovered) the affected service is stored in $updated$ to recompute its new aggregated QoS. Finally, for each service $w \in updated$, we recompute its aggregated QoS using the updated values of each affected input. If the QoS has been improved, the service is added to the queue to expand it later.

```
1:  function QoS-UPDATE(G_S = {V, E})
2:      /*qos is a table indexed by inputs (i)
3:      associated to their aggregated QoS (q)*/
4:      qos[i, q] ← []
5:      for all i ∈ I, I ⊂ V do
6:          qos[i] ← φ
7:      queue ← So
8:      while queue ≠ ∅ do
9:          /* Queue sorted by aggregated QoS */
10:         w ← POP(queue)
11:         updated = {}
12:         for all o_w ∈ Out_w do
13:             for all i_w' ∈ ch_G(o_w) do
14:                 if V_Q(w) ≺ qos[i_w'] then
15:                     qos[i_w'] ← V_Q(w)
16:                     updated ← updated ∪ w'
17:         for all w ∈ updated do
18:             if cost w has been improved then
19:                 queue ←INSERT(w, queue)
20:     return qos
```

Fig. 4. Dijkstra-based algorithm to compute the best QoS for each input and output in the *Service Match Graph* G_S.

5.3 Graph optimizations

Finding the composition with the minimum number of services is a very hard combinatorial problem which, in most cases, has a very large search space, mainly determined by the size of the *Service Match Graph*. In order to improve the scalability with the number of services, we apply a set of admissible optimizations to reduce the search space. At each pass, the algorithm analyzes different criteria to identify services that are redundant or can be substituted by better ones, so the size of the graph decreases monotonically. The different passes that are sequentially applied are: 1) elimination of services that do not contribute to the outputs of the request; 2) pruning of services that lead to suboptimal QoS; 3) combination of interface (inputs/outputs) and QoS equivalent services; and 4) replacement of interface and QoS dominated services. These optimizations are an extension of the optimizations presented in [30] to support QoS.

The **first pass** selects the set of reachable services in the *Service Match Graph*. Starting from the inputs of Si, it selects all those services whose outputs match any inputs of Si. This step is repeated with the new services until the empty set is selected. Those services that were not selected do not contribute to the expected outputs of the composition and can be safely removed from the graph.

The **second pass** prunes the services of the graph that are suboptimal in terms of QoS, i.e., they cannot be part of any optimal QoS composition. To do so, we compute the maximum admissible QoS bound for each input in the graph. In a nutshell, the maximum bound of the inputs of a service w can be calculated by selecting the maximum QoS bound among the bounds of all inputs matched by the outputs of the service w and subtracting the QoS of w. This can be recursively defined as:

$$max^i_Q(i_w) = \\ = \begin{cases} V_Q(w) \ominus F_Q(w) & \text{if } Out_w = \emptyset \\ \max\limits_{\forall o_w, \forall i_{w'} \in ch_G(o_w)} (max^i_Q(i_{w'})) \ominus F_Q(w) & \text{if } Out_w \neq \emptyset \end{cases}$$

The value of max^i_Q for each input in the graph can be easily calculated by propagating the bounds from Si to So. For example, in Fig. 1, we start computing the maximum bound of the inputs of Si (*xsd:boolean*). Since Si has no outputs, max^i_Q(*xsd:boolean*) is calculated as $V_Q(Si) \ominus F_Q(Si) = 410\ ms - 0\ ms$. Then, we select all the services whose outputs match *xsd:boolean*. In this case there is just one service, *ML Predictor Service*. The bounds of its inputs are now computed by subtracting out the F_Q(*ML Predictor Service*) from the maximum bound of the inputs that this service matches. Since there is just one input matched (*xsd:boolean* from Si) whose bound is *410 ms*, we have $max^i_Q(i) = 410ms - 210ms = 200ms$ for each input i of the service. In the next step, we have three services that match the new calculated inputs (*Free Geoloc Service*, *Premium Geoloc Service* and *Transaction Service*). The maximum bounds of the inputs of these services are $200ms - 180ms = 20ms$, $200ms - 40ms = 160ms$ and $200ms - 130ms = 70ms$ respectively. Note that, since the maximum bound of *Transaction Service* is *70ms*, the service *Secure Payment* is out of the bounds (its output QoS is *80 ms*), so it can be safely pruned.

The **third and the forth pass** analyze service equivalences and dominances in the *Service Match Graph*. It is very frequent to find services from different providers that offer similar services with overlapping interfaces (inputs/outputs). In scenarios like this, it is easy to end up with large *Service Match Graph* that make very hard to find optimal compositions in reasonable time. One way to reduce the complexity without losing information is to analyze the interface equivalence and dominance between services in order to *combine* those that are equivalent, or replace those that are dominated in terms of the interface they provide and the QoS they offer. In a nutshell, we check three objectives to compare services: the *amount* of information they need to be invoked (inputs), the *amount* of information they return (outputs), and their QoS. If a set of services are equal in all objectives, they are equivalent and they can be combined into an abstract service with several possible implementations. If a service is equal in all objectives and at least better in one objective (it requires less information to be invoked, produces more information or has a better QoS), then the service *dominates* the other service. A more detailed description of the interface and dominance optimizations is described in [30].

Note that optimizations are applied right before all semantic matches are computed in the *Service Match Graph*, since the optimizations are based on the analysis of the I/O matches among services. For this reason, they cannot be applied during the calculation of the graph (this would require to precompute in advance missing relations during the graph generation, which does not provide any benefit as this is what the *Service Match Graph* generation algorithm already does). On the other hand, optimizations are applied sequentially to save computation time, since the number of services in the graph decreases monotonically in each step. In order to take advantage of this, faster optimizations are applied first so that the slower optimizations in the pipeline can work with a reduced set of services.

5.4 Hybrid algorithm

Each service in the composition graph may have different services that match each input, thus there may exist multiple combinations of services that satisfy the composition request with the same or different QoS. The goal of the hybrid search is to extract good solutions from the composition graph, optimizing the total number of involved services in the composition and guaranteeing the optimal QoS. Thus, for each input we select just one service of the graph to match that input, until the best combination is found. The hybrid search performs a local search to extract a good solution and in the remaining time, it tries to improve the solution by running a global search.

Fig. 5 shows the pseudocode of the **local search** strategy. The algorithm starts with a composition graph, the inputs of the service Si marked as unresolved (the expected outputs of the request) and the service Si selected to be part of the solution. An *unresolved input* is an input that can be matched by many different outputs but no decision has been made yet. Using the list of the unresolved inputs to be matched, the method RANK-RESOLVERS returns a list of services that match any of the unresolved inputs. Services are ranked according to the number of unresolved inputs that match, so the service that matches more inputs is considered first to be part of the solution. Then, for each input that the selected service can match, the method CYCLE performs a forward search to check if resolving the selected input with that service leads to a cycle. For example, in Fig. 3, if we select the service K to match the input of I after having decided to resolve the input of K with the service I, we end up with an invalid composition, so K is an invalid resolver for I and it must be discarded. Once all resolvable inputs are collected in $resolved$, the method RESOLVE creates a copy of the current graph where the inputs in $unresolved$ are matched only by the selected service, i.e., any other match between any output from a different service to that input is removed from the graph. If the selected service was not already selected, then all its inputs are then marked as unresolved and a recursive call to LSBT is performed to select a new service to resolve the remaining inputs, until a solution is found. If a dead end is reached (a solution that has no services to resolve the remaining inputs without cycles)

```
1:  function LOCAL-SEARCH(G_S = {V, E})
2:      return LSBT(G_S, In_Si, {Si})
3:
4:  function LSBT(G_S, unresolved, services)
5:      if unresolved = ∅ then return G_S
6:      servs ← RANK-RESOLVERS(unresolved)
7:      for each w ∈ servs do
8:          resolved ← {}
9:          matched ← Out_w ⊗ unresolved
10:         for each input ∈ matched do
11:             if ¬CYCLE(G_S, w, input) then
12:                 resolved ← resolved ∪ input
13:         if resolved ≠ ∅ then
14:             unresolved ← unresolved \ resolved
15:             if w ∉ services then
16:                 unresolved ← unresolved ∪ In_w
17:             G'_S ← RESOLVE(G_S, w, resolved)
18:             services ← services ∪ w
19:             result ← LSBT(G'_S, unresolved, services)
20:             if result ≠ fail then return result
        return fail
```

Fig. 5. Local search algorithm to extract a composition from a graph.

```
1:  function CYCLE(G_S = {V, E}, w, i_w')
2:      W_visited ← {w'}
3:      W_new ← {w'}
4:      while W_new ≠ ∅ do
5:          W_reached ← {}
6:          for all w_n ∈ W_new do
7:              for all o_wn ∈ Out_wn do
8:                  for all i_w'n ∈ ch_GS(o_wn) do
9:                      if d⁻_GS(i_w'n) = 1 ∧ w'_n ∉ W_visited then
10:                         if w'_n = w then return true
11:                         W_reached ← W_reached ∪ w'_n
12:         W_new ← W_reached
13:         W_visited ← W_visited ∪ W_new
14:     return false
```

Fig. 6. Näive breadth-first-search algorithm to check whether using the service w to resolve the input $i_{w'}$ of a service w' leads to a cycle.

the algorithm backtracks to a previous state to try a different service (L.7).

An implementation of the *CYCLE* method is provided in 6. The algorithm performs a *look-ahead* check in a breadth-first fashion to determine whether matching the selected input i with an output of the service w leads to a cyclic dependency. This is done by traversing only the resolved matches, i.e., inputs that are matched by just one output of a service, until the selected service w is reached, proving the existence of a cycle. A more memory efficient implementation of the cycle algorithm can be done using the *Tarjan's strongly connected components* algorithm [38], stopping at the first strongly connected component detected.

After the local search is used to find a good solution, the **global search** is performed in the remaining time to obtain a better solution by exhaustively exploring the space of possible solutions. In a nutshell, this algorithm works as follows: Given a *Service Match Graph* G_S, with some unresolved inputs, which initially are the inputs of the service Si, the algorithm selects an input to be resolved and for each service candidate that can be used to resolve that input, it generates a copy of the graph G_S but with the input resolved (i.e., the selected service is the only one that matches the unresolved input). The algorithm enqueues each new graph to be expanded again, and repeats the process by extracting the graph with the minimum number of services from the queue, until it eventually finds a graph with no unresolved inputs.

Fig. 7 shows the pseudocode of the global search algorithm. The algorithm starts computing the optimal QoS of the graph with the method *QoS-UPDATE*. This method returns a key-value table $qos[i, q]$ where each key corresponds with an input i of the graph, and each value q its optimal aggregated QoS $q = V_Q^{in}(i)$. Then, the inputs of the service Si of the graph are added to I_{un} to mark them as unresolved (L.8). In order to minimize the number of possible candidates for each unresolved input, we compute and propagate a range of valid QoS values, called QoS bounds, and defined as an interval $[min, max]$. These bounds determine the range of valid accumulated QoS values of the outputs that can be used to match each of the unresolved inputs without exceeding the optimal end-to-end QoS of the final composition. The min value is the optimal QoS for the input, i.e., there is no output in the graph that can match the input with a lower QoS, whereas the max value is the maximum QoS value supported. If this bound is exceeded, the total aggregated QoS of the composition worsens. For example, in Fig. 1, the bounds of the input *ont4:ClientID* of the service *Premium Geoloc Service* are $[20ms, 160ms]$. If we exceed the min bound (*20 ms*), the output QoS of the service gets worse ($> 60ms$), which also affects the optimal QoS of the input *ont1:Location*. However, as long as the max bound is not exceeded ($\leq 160ms$), the optimal accumulated QoS of the *ML Predictor Service* would not be affected.

The method *COMPUTE-V_Q* is used to compute the value of the V_Q function (Eq. 1) using the best QoS values of inputs, stored in qos ($qos[i] = V_Q^{in}(i)$). A tuple $\langle G_S, I_{un}, qos, W_{sel}\rangle$, where G_S is the current graph, I_{un} are the unresolved inputs of G_S, qos is the best aggregated QoS values for each input in G_S and W_{sel} is the set of the selected services, defines the components of a partial solution. Each partial solution is stored in a priority queue, which is sorted by the number of services W_{sel}. This allows an exploration of the search

space in a breadth-first fashion, so the solution with the minimum number of services is always expanded first. At each iteration, a partial solution is extracted from the queue to be refined (L.12). If the partial solution has no unresolved inputs, the solution is complete, and has the minimum number of services. If the partial solution still has some unresolved inputs, it is refined by selecting an unresolved input with the method *SELECT*. This method selects the input to be resolved, using a *minimum-remaining-values* heuristic. This heuristic selects always the input with less resolvers (services candidates) in order to minimize the branching factor. The list of services that can match the selected input with a total aggregated QoS value within the $[min, max]$ bound is calculated with the method *RESOLVERS*. For each valid service, the algorithm performs a *look-ahead* search to check whether using the current service to resolve the selected input leads to an unavoidable cycle. If so, the service is prematurely discarded to save computation time and space. If it does not lead to a cycle, then a copy of the graph (G'_S) with the selected input resolved is generated, and the input is also removed from the set of unresolved inputs. Using the optimal aggregated QoS values for the inputs of the graph, stored in qos, the algorithm computes the aggregated QoS value of the service w. If this value is worse than the min bound (COMPUTE-$V_Q(w, qos') \succ min$), then the aggregated QoS value of some inputs and outputs of the graph may be affected. Thus, a repropagation of the QoS values for each input and output is computed again over the new graph G'_S (L.22). For example, if the *Business Service Info* increments its response time to *40 ms*, a repropagation is required to recompute the accumulated QoS of all the services that may be affected. In this case, the *Premium Geoloc Service* increments its accumulated QoS cost from *60 ms* to *80 ms*, as well as the optimal QoS of the *ont1:Location*.

Finally, if the current service is not part of the current solution, its inputs are added to the unresolved table, and a new bound for each input is computed. The min bound corresponds with the optimal value, which is stored in qos'. In order to compute the max bound, we need to *subtract* the QoS of the selected service ($F_Q(w)$) from the max bound of the resolved input, using the operator $\ominus$ (L.25). This new partial solution is inserted in the queue to be expanded later on.

6 Evaluation

In order to evaluate the performance of the proposed approach, we conducted two different experiments. In the first experiment, we evaluated the approach using the datasets of the Web Service Challenge 2009-2010 [39]. The goal of this first experiment was to evaluate the peformance and scalability of the proposed approach on large-scale service repositories. In the second experiment, we tested the algorithm with five random datasets in order to better analyze the differences of the performance

```
1:  function GLOBAL-SEARCH(G_S)
2:      qos[i,q] ← QoS-UPDATE(G_S)
3:      max ← COMPUTE-V_Q(Si, qos)
4:      W_sel ← {Si}
5:      /* I_un is a key-value table where the keys are
6:      unresolved inputs and the values their QoS bounds */
7:      for i_Si ∈ In_Si do
8:          I_un[i_Si] ← [qos[i_Si], max]
9:      /* Queue sorted by |W_sel| */
10:     queue ← INSERT(⟨G_S, I_un, qos, W_sel⟩, queue)
11:     while queue ≠ ∅ do
12:         ⟨G_S, I_un, qos, W_sel⟩ ← POP(queue)
13:         if I_un = ∅ then return G_S
14:         input ← SELECT(I_un)
15:         [min, max] ← I_un[input]
16:         for all w ∈ RESOLVERS(input, [min, max]) do
17:             if ¬CYCLE(G_S, w, input) then
18:                 G'_S ← RESOLVE(G_S, w, {input})
19:                 I'_un ← REMOVE(i, I_un)
20:                 qos' ← qos
21:                 if COMPUTE-V_Q(w, qos') ≻ min then
22:                     qos' ← QoS-UPDATE(G'_S)
23:                 if w ∉ W_sel then
24:                     W'_sel ← W_sel ∪ w
25:                     max' ← max ⊖ F_Q(w)
26:                     for i_w ∈ In_w do
27:                         min' ← qos'[i_w]
28:                         I'_un[i_w] ← [min', max']
29:             queue ← INSERT(⟨G'_S, I'_un, qos', W'_sel⟩, queue)
        return fail
```

Fig. 7. Global search algorithm to extract the optimal composition.

between the local and the global search. All tests were executed with a time limit of 5 min. Solutions produced by our algorithm are represented as *Service Composition Graphs* (no BPEL was generated).

6.1 Web Service Challenge 2009-2010 datasets

The datasets of the Web Service Challenge 2009-2010 range from 572 to 15,211 services with two different QoS properties: response time and throughput. Table 2 shows the results obtained for each dataset and for each QoS property. The response time is the average time (measured in milliseconds) that a service takes to respond to a request. The throughput, as defined in the WSC, is the average ratio of invocations per second supported by a service.

Row #*Graph services* shows the number of services of the composition graph and #*Graph services (opt)* the number of services after applying the graph optimizations. As can be seen, the optimizations reduce, on average, by 64% the number of services in the initial composition graph. This indicates that equivalence and dominance analysis of the QoS and the functionality of services is a

TABLE 2
Validation with the WSC 2009-2010

		D-01	D-02	D-03	D-04	D-05
#Services in the dataset		572	4,129	8,138	8,301	15,211
Validation with Response Time						
Optimal Response Time (ms)		500	1,690	760	1,470	4,070
#Graph services		81	141	154	331	238
#Graph services (opt)		21	57	15	160	126
Local Search	#Services	5	20	10	40	32
	Time (s)	0.613	0.988	2.608	7.767	2.920
Global Search	#Services	5	20	10	-	32
	Time (s)	0.617	1.580	2.613	-	24.971
Validation with Throughput						
Optimal Throughput (inv/s)		15,000	6,000	4,000	4,000	4,000
#Graph services		81	141	154	331	238
#Graph services (opt)		10	43	90	156	69
Local Search	#Services	5	20	15	62	31
	Time (s)	0.343	1.173	1.933	8.571	2.562
Global Search	#Services	5	20	10	-	30
	Time (s)	0.345	1.246	2.085	-	119.322

TABLE 3
Comparison with the top 3 WSC 2010

		R.Time	Through.	Min. Serv.	Time (ms)
D-01	CAS [4]	500	15,000	**5**	78
	RUG [6]	500	15,000	10	188
	Tsinghua [5]	500	15,000	9	109
	Our approach	500	15,000	**5**	956
D-02	CAS [4]	1,690	6,000	**20**	94
	RUG [6]	1,690	6,000	40	234
	Tsinghua [5]	1,690	6,000	36	140
	Our approach	1,690	6,000	**20**	2,171
D-03	CAS [4]	760	4,000	**10**	78
	RUG [6]	760	4,000	11	234
	Tsinghua [5]	760	4,000	18	125
	Our approach	760	4,000	**10**	4,693
D-04	CAS [4]	1,470	4,000	73	156
	RUG [6]	1470	4,000	133	390
	Tsinghua [5]	1,470	4,000	133	188
	Our approach	1,470	4,000	**40**	16,338
D-05	CAS [4]	4,070	4,000	32	63
	RUG [6]	4,070	4,000	4,772	907
	Tsinghua [5]	4,070	4,000	4,772	531
	Our approach	4,070	4,000	**30**	122,242

powerful technique to reduce the search space in large scale problems. Rows *Local search* and *Global search* show the number of services of the solution obtained with each respective method as well as the total amount of time spent in the search. The global search found the best solution for each dataset and for each QoS property, except for the dataset 04, where the composition with the minimum number of services could not be found due to combinatorial explosion. However, in those cases, the local search strategy is able to find an alternative solution very fast. Note also that, in many cases, the local search obtains the best solution (comparing it with the global search) except for the throughput in datasets 03 and 05.

We have compared our approach with the top-3 of the Web Service Challenge 2010 [40]. Table 3 shows this comparison following the same format and the same rules of the Web Service Challenge. The format, rules and other details of the challenge are described in [40]. Third and forth columns show the response time and the throughput obtained for each dataset. Note that, since all these algorithms minimize a single QoS, these values are computed by executing the algorithm twice, one for each QoS. Unfortunately, the results provided by the WSC organization in [40] show only the minimum number of services for both executions (fifth column). Thus, the number of services obtained for both the response time and throughput is unknown, which makes it hard to compare with our results. Even so, using the same evaluation criteria, our approach obtains the optimal QoS for the response time and the throughput, and also improves the number of services in D-04 (40 vs 73) and D-05 (30 vs 32) with respect to the solutions obtained by the winner of the challenge (the minimum number of services obtained for each dataset is highlighted). The last column shows the total execution time of each algorithm. The total time includes the time spent to obtain the solution for the response time and for the throughput.

Our approach takes, in general, more time to obtain a solution. However, it should be noted that we show the best results achieved by the hybrid approach, i.e., if the global search improves the solution of the local search, we show that solution along with the time taken by the global search. Anyway, the local search always provide a first good solution very fast. For example, as can be seen in Table 2, the optimal solution for D-05 has 30 services and has been obtained in 119.322 s, but the local search obtained a solution with 31 services in 2.56 s, still better than the solution with 32 services obtained by [4] (Table 3). Moreover, it should also be noted that the problem of finding the optimal composition with minimum number of services and optimal QoS is much harder than just optimizing the QoS objective function, which is the problem solved by the participants of the WSC 2010. Although the problem is intractable and requires exponential time, it can be optimally solved for many particular instances in a reasonable amount of time using adequate optimizations even in large datasets as shown in Tables 2 and 5. This is one of the main reasons why a combination of a local and global search can achieve good results in a wide variety of situations, in contrast with pure greedy strategies or with pure global optimization algorithms.

We also compare the results obtained with Chen et al. [7], who offer a detailed analysis of their results. This comparison is shown in Table 4. Solutions are compared according to their QoS and number of services. A solution is better if 1) its overall QoS is better or 2) has the same QoS but less services. The results show that our

algorithm always gets same or better results. Concretely, it finds solutions with optimal QoS and less services in D-01, D-02, D-04 and D-05 (response time), and D-03 (throughput). It also finds a solution with a better QoS (4000 inv/s vs 2000 inv/s) in D-04 (throughput).

TABLE 4
Detailed comparison with [7]

		D-01	D-02	D-03	D-04	D-05
Chen et al.	R. Time	500	1,690	760	1,470	4,070
	Services	8	21	**10**	42	33
Our approach	R. Time	500	1,690	760	1,470	4,070
	Services	**5**	**20**	**10**	**40**	**32**
Chen et al.	Throughput	15,000	6,000	4,000	2,000	4,000
	Services	**5**	**20**	21	40	**30**
Our approach	Throughput	15,000	6,000	4,000	**4,000**	4,000
	Services	**5**	**20**	**10**	62	**30**

6.2 Randomly generated datasets

Although the global search is able to obtain solutions with a lower number of services, a first look at the results with the WSC dataset might suggest that the difference of both strategies is not very significant, as most of the obtained solutions have the same number of services. However, this may be due to a bias in the repository, since all the datasets of the WSC are generated using the same random model. In order to better evaluate and characterize the performance of the hybrid algorithm, we generated a new set of five random datasets that range from 1,000 to 9,000 services. These datasets are available at https://wiki.citius.usc.es/inv:downloadable_results:ws-random-qos. Table 5 shows the solutions obtained.

TABLE 5
Validation with random datasets

		R-01	R-02	R-03	R-04	R-05
#Services in the dataset		1,000	3,000	5,000	7,000	9,000
Validation with Response Time						
Optimal Response Time (ms)		1,430	975	805	1,225	1,420
#Graph Services		54	168	285	383	499
#Graph Services (opt)		22	50	54	56	99
Local Search	#Services	7	18	20	15	19
	Time (s)	0.183	0.403	0.422	0.515	0.641
Global Search	#Services	7	14	15	15	16
	Time (s)	0.243	0.767	4.088	0.740	3.131
Validation with Throughput						
Optimal Throughput (inv/s)		1,000	2,500	1,500	2,000	2,500
#Graph Services		54	168	285	383	499
#Graph Services (opt)		19	46	133	116	103
Local Search	#Services	7	17	24	19	23
	Time (s)	0.072	0.143	0.606	0.732	0.450
Global Search	#Services	7	12	12	15	16
	Time (s)	0.155	0.310	2.479	1.485	1.714

We found that in these datasets, the solutions obtained with the global search strategy are, on average, $\approx 16\%$ smaller than the ones obtained with the local search, whereas the differences in seach time are less pronounced than in the previous experiment. These findings suggest that the performance of each strategy highly depends on the underlying structure of the service repository, which is mostly determined by the number of services and the existing matching relations.

In order to test whether these differences are statistically significant or not, we conducted a nonparametric test using the *binomial sign test* for two dependent samples with a total of 20 datasets (5 WSC w/response time + 5 WSC w/throughput + 5 Random w/response time + 5 Random w/throughput). The null hypothesis was rejected with *p-value* ≈ 0.01 [41], meaning that both strategies (local and global search) find significantly different solutions. Thus, a hybrid strategy can perform better in many different scenarios, since it achieves a good tradeoff between quality and execution time.

This evaluation shows that, on one hand, the combination of local and global optimization is a general and powerful technique to extract optimal compositions in diverse scenarios, as it brings the best of both worlds. This is specially important when only a little or nothing is known concerning the structure of the underlying repository of services. On the other hand, the results obtained with the Web Service Challenge 2009-2010 show that the hybrid strategy performs better than the state-of-the-art, obtaining solutions with less services and optimal QoS.

7 Conclusions

In this paper we have presented a hybrid algorithm to automatically build semantic input-output based compositions minimizing the total number of services while guaranteeing the optimal QoS. The proposed approach combines a set of graph optimizations and a local-global search to extract the optimal composition from the graph. Results obtained with the Web Service Challenge 2009-2010 datasets show that the combination of graph optimizations with a local-global search strategy performs better than the state-of-the-art, as it obtained solutions with less services and optimal QoS. Moreover, the evaluation with a set of randomly generated datasets shows that the hybrid strategy is well suited to perform compositions in diverse scenarios, as it can achieve a good tradeoff between quality and execution time.

Appendix A
Computational Complexity

The calculation of the optimal QoS can be computed in polynomial time for a given *Service Match Graph* using classical shortest path algorithms such as Dijkstra or Bellman-Ford. But, as stated in the introduction, there can exist multiple solutions with the same global QoS but different number of services. Thus, in many scenarios, optimizing the QoS objective function is not enough to

provide the best possible answer. However, it turns out that optimizing the number of services of a composition is an intractable problem. The next theorem proves that the Service Minimization Problem (SMP) is a NP-Hard combinatorial optimization problem.

Theorem. *Finding the minimum number of services whose outputs match a given set of unresolved (unmatched) concepts is a NP-Hard combinatorial optimization problem.*

Proof: We will show that the Service Minimization Problem (SMP) is NP-Hard by proving that the optimization version of the Set Cover Problem (SCP), a well-known NP-Hard problem, is polynomial-time *Karp* reducible to SMP $SCP \leq_P SMP$. The optimization version of the SCP problem is defined as follows: given a set of elements $U = \{u_1, \ldots, u_m\}$ and a set S of subsets of U, find the smallest set (cover) $C \subseteq S$ of subsets of S whose union is U. The decision version of this problem, stated as that of deciding whether exists a cover C_{SCP} of size k or less ($|C_{SCP}| \leq k$), is NP-Complete. We will also consider the simplest form of the SMP that can be contained in a *Service Match Graph*, which is defined as follows: given a service w_U and a set of candidate services $W_S = \{w_1, \ldots, w_n\}$ such that $O_{w_1} \otimes I_{w_U} \neq \emptyset \wedge \cdots \wedge O_{w_n} \otimes I_{w_U} \neq \emptyset$, select the smallest subset of services from W_S such that the union of the outputs of the services from W_S, O_{W_S}, satisfies $O_{W_S} \otimes I_{w_U} = I_{w_U}$, i.e., the outputs of the services contained in W_S match all the inputs of w_U. As in the SCP, the decision version of this optimization problem is defined as that of deciding whether exists a subset of candidate services C_{SMP} of size k or less ($|C_{SMP}| \leq k$) such that the union of the outputs of the services in C_{SMP} match all the inputs of w_U.

In order to prove that the SMP optimization problem is NP-Hard, we need to demonstrate that its corresponding decision problem is NP-Complete. We will therefore reduce the SCP problem by means of a function φ that transforms any arbitrary instance of the SCP into an instance of the SMP in polynomial time. We have to prove that 1) $\varphi(U,S)$ is a SMP problem; 2) φ runs in polynomial time; and 3) there is a set covering of $\varphi(U,S)$ of size k or less if and only if there is a set covering of U in S of size k or less.

Given a pair (U,S), we define $\varphi(U,S) = (w_U, W_S)$ such that:

- $w_U = \{I_{w_U} = U = \{u_1, \ldots, u_n\}, \emptyset\}$, where u_i is the ith unresolved input of w_U.
- $\forall s_i = \{u_{i_1}, \ldots, u_{i_n}\} \in S$, $\exists w_i \in W_S$ such that $w_i = \{\emptyset, O_{w_i}\}$ and $O_{w_i} \otimes I_{w_U} = s_i$

By this definition, the $\varphi(U,S)$ maps each element $u \in U$ to an input of the service w_U. Each subset $s_i \in S$ is also mapped to a service whose outputs match exactly the inputs of w_U that correspond with the elements of s_i. This mapping can be computed by adding a match from an arbitrary output of each service $w_i \in W_S$ to each input $u_i \in s_i$, which clearly runs in linear time in the size of U. Moreover, $\varphi(U,S)$ is a Service Minimization Problem according to its definition.

Now suppose there is a set covering $|C| \leq k, C \subseteq S$ of U. Thus, $\forall u \in U, \exists c_i \in C$ such that $u \in c$. From the services (w_U, W_S) constructed from (U,S) by $\varphi(U,S)$, there exists $w_i \in W_S$ such that $O_{w_i} \otimes I_{w_U} = c_i \subseteq I_{w_U}$, and so $\bigcup_i (O_{w_i} \otimes I_{w_U}) = I_{w_U} = C$, i.e., the outputs of the services from the set W_S of size k or less represent a cover of the Service Minimization Problem $\varphi(U,S)$. □

APPENDIX B
ALGORITHM ANALYSIS AND DISCUSSION

The proposed approach consists of a hybrid algorithm that optimizes both the global QoS and selects the composition with the minimum number of services that preserves the optimal QoS. As demonstrated in Appendix A, the problem of minimizing the number of services is NP-Hard. Thus, under the $P \neq NP$ assumption, there is no polynomial time algorithm that can exactly solve this optimization problem. However, although it is in general intractable, in practice many instances of the problem, as shown in the evaluation section, can be optimally solved in reasonable time. In those situations, it may be preferable to provide optimal solutions instead of just sub-optimal ones. Our approach takes advantage of a hybrid strategy that combines a local search and a global search plus the use of preprocessing optimizations and search optimizations (minimum-remaining-values heuristic, cycle detection, QoS bounds propagation) in order to achieve a good trade-off between optimality of the solution and computation time. Here we analyze the complexity of the proposed techniques.

B.1 Cycle detection

The cycle detection is implemented as a Look-Ahead strategy, that traverses all the resolved matches, starting from the current service (the one selected to resolve a new unresolved input), until no more services are reachable. This strategy seeks to discover whether the current service is a valid candidate or not by checking if it can lead to a dependency cycle, so it can be prematurely discarded. The cycle detection algorithm takes $O(|V|+|E|)$, since every service, input, output and match between inputs and outputs have to be traversed in worst-case.

B.2 QoS Update

The QoS update method calculates the optimal end-to-end QoS through the graph. This method is also used to recalculate optimal QoS bounds whenever a local QoS bound is excedeed. This problem can be modeled as a shortest path problem with generalized costs for QoS (as shown in Section 4.3) and solved using Dijkstra's algorithm. The worst-case time complexity of this algorithm

is as follows: given a *Service Match Graph* $G_S = (V, E)$, where $W_R \subset V$ is the set of services in the graph, there are at most $|W_R|$ calls to POP method to extract the lowest scored service from the queue. Since the queue is implemented as a binary heap, the *POP* and *INSERT* methods have a time of $O(log(n))$, where n is the size of the queue. Thus, in the worst case, the running time is $O(|W_R| \cdot log(|W_R|))$, plus the (at most) $|E|$ updates of neighbor services that are reinserted into the queue. Therefore, the overall time is $O((|E| + |W_R|) \cdot log(W_R))$.

B.3 Local search

This method performs a heuristically guided local search to minimize the number of services of the optimal end-to-end QoS composition. At each step, it selects the most promising candidate by selecting the one with fewer inputs that matches the largest number of unresolved inputs. If the algorithm gets stuck at some point, i.e., it reaches a point where no service can be selected without leading to a cyclic dependency, it backtracks to try the next most promising candidate service. The algorithm calls *RANK-RESOLVERS* to rank the candidates according to the number of unresolved inputs that each candidate can match and, in case of draw the service with less inputs is preferred. The sorting of services takes $O(n \cdot log(n))$ using merge sort, where n is the number of services. Each time a service is selected, the method *RESOLVE* creates an updated copy of the graph in $O(|V| + |E|)$.

Assumming non-cyclic dependencies in the *Service Match Graph*, in the worst case the algorithm have to select all the services from the graph until no unresolved inputs are left. Thus, in the first step $t_{|W_R|}$ the algorithm ranks all the $|W_R|$ services in $O(|W_R| \cdot log(|W_R|))$, selects the first one and generates a new copy of the graph in $O(|V| + |E|)$. The running time of this step is $O(|W_R| \cdot log(|W_R|) + O(|V| + |E|) = O(|W_R| \cdot log(|W_R|))$. In the next step $t_{|W_R|-1}$, the algorithm ranks $|W_R| - 1$ services, selects the best one, creates a copy of the graph and so on. Therefore, the asymptotic upper bound of the running time of $t_{|W_R|} + t_{|W_R|-1} + \cdots + t_1$ is $O(|W_R| \cdot log(|W_R|))$.

In the absence of the assumption of non-cyclic dependencies, the asymptotic upper bound analysis shows that the time complexity grows exponentially with the depth of the search, since in the worst-case the algorithm fails (backtracks) at each step until the last combination of services is explored. However, in practice, this upper bound seems far from the average-case. As shown in the evaluation (Section 6), the growth of the time with respect to the size of the graph is closer to the best-case scenario, since an exponential number of backtracks due to cylic dependencies is extremely rare. In any case, the algorithm can be easily adapted to perform better in the worst-case scenario, for example by limiting the number of candidates to the top-*K* best services for each unresolved input.

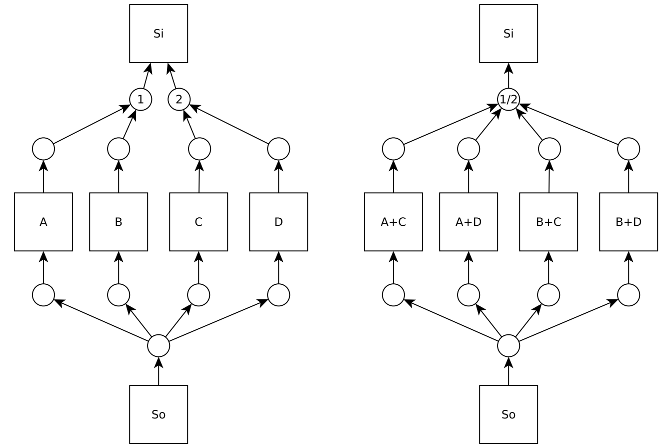

Fig. 8. Reduction of the left graph into the right graph by computing all possible combinations of services

B.4 Global search

The aim of the global search algorithm is to perform an exhaustive search to find the minimum combination of services that satisfy the composition request with optimal QoS. The algorithm explores every possible valid combination of services in a breadth-first fashion by resolving one input at a time. For each unresolved input with $k > 1$ candidates, new k different states are created by calling the *RESOLVE* method and pushed to the queue for further expansion. In order to calculate an asymptotic upper bound for the time complexity, we can compute the number of combinations of services that the algorithm needs to extract from the queue in the worst-case. To this end, we first count the maximum number of combinations (solutions) that we can generate for a simple graph with fixed size and then we generalize the problem for a graph of any size.

Left graph from Figure 8 shows an example of a *Service Match Graph* with 4 services (excluding Si and So). As can be seen, Si requires two inputs, 1 and 2. On the other hand, the outputs of A and B match the input 1 whereas the outputs of services C and D match the input 2. Therefore, in order to match both inputs, we can select services A and C, A and D, B and C or B and D (2×2 combinations). By computing all possible combinations, we can reduce the graph from the left, where Si has two inputs, to the graph from the right, where Si has just one input.

In general, given a service w with $|I_w| = k$ inputs and $c_1, c_2, \ldots, c_k$ set of candidate services for each input, there are $\prod_i |c_i|$ combinations of services, i.e., we can replace the k inputs with k sets of candidate services by one input with $\prod_i |c_i|$ candidates. Since each service can have in turn some inputs with other candidates, we can recursively replace each service with all the possible combinations of services that can be generated. This process leads to a flattening of the graph until there

is just one level with all the possible combinations of services (compositions) that can be generated for a given *Service Match Graph*. Thus, the problem of counting the number of possible solutions in the worst-case can be reduced to the following: given a *Service Match Graph* with $|W_R|$ services, what is the maximum of products of partitions of W_R? More formally, given a set S ($|S| \geq 1$), choose n partitions $c_1, c_2, \ldots, c_n$ such that $\sum_i |c_i| = |S|$ and $\prod_i |c_i|$ is maximized. For example, given 11 services, we can take 3 groups of 3 services and one with the remaining 2 services, so the product of the partition is $3^3 \cdot 2 = 54$, which is the maximum. Finding an upper bound for this value will gives us an upper bound for the maximum number of compositions that can be enumerated in the worst-case, i.e., for the most complex *Service Match Graph* that can be generated with $|W_R|$ services. It can be proved that, for any set of size n, the maximum can be obtained by partitioning the set into groups of 2 and 3 elements, with no more than 2 groups of 2 elements. From this it follows that the maximum product is bounded by $3^{n/3}$, so we can conclude that $O(3^{n/3})$ is a tight asymptotic upper bound on the running time in the worst-case.

However, it should be noted that although the calculation of an optimal solution for the problem in the worst-case requires exponential time with the size of the graph, in practice, the number of services for a particular request is usually orders of magnitude lower that the number of available services in the dataset (see Table 2 and 4). In addition to this, the optimizations introduced in Section 5.3 plus the global QoS bound propagation, the minimum-remaining-values heuristic and cycle detection used in the global search are aimed to reduce further the size of the explored search space by decreasing the number of analyzed services.

Appendix C
Differences with previous work

In [30] we presented an integrated approach for discovery and composition of semantic Web services. However, the framework does not include any of the novelties that are presented in this approach. Our previous work presents an integrated framework for automatic I/O driven discovery and composition of semantic Web services and analyzes the impact of the discovery in the whole process, but with no QoS support. In contrast, in this work we present a hybrid composition algorithm that optimizes both QoS and the number of services, which is a different and a harder problem. The main differences are:

- The *Service Model* has been extended to give support for QoS properties.
- The computation of the *Service Match Graph* for this problem is different. In this work, all the semantic matches between all the services in the graph are computed in order to be able to guarantee an optimal end-to-end QoS. However, in [30], the *Service Match Graph* contains only the matches from the outputs of previous layers to the inputs of subsequent layers, i.e., the inputs of a service that appears in the ith layer can be matched only by the outputs of services that are in any jth layer where $j \in [0, i-1]$. This condition is enough to find the smallest composition (in terms of number of services and length of the composition) but it is not enough to guarantee the optimal QoS since there are missing relations that can be part of the optimal solution.
- *Service Match Graph* optimizations have been extended to take into account QoS. Also, a new step in the optimization pipeline has been included to prune suboptimal QoS services (i.e., services that cannot be part of the optimal solution).
- The proposed composition algorithm is completely different. The algorithm from [30] is focused on the minimization of Web services using an A* algorithm with admissible state-space pruning. However, this technique is not enough to cope with the complexity of this new problem at large scale. Thus, we developed a new algorithm which consists of a hybrid strategy to optimize both global end-to-end QoS and the number of services, which is a different and also a harder problem.

Acknowledgment

This work was supported by the Spanish Ministry of Economy and Competitiveness (MEC) under grant TIN2014-56633-C3-1-R and the Galician Ministry of Education under the project CN2012/151. Pablo Rodríguez-Mier is supported by an FPU Grant from the MEC (ref. AP2010-1078).

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services*, ser. Data-Centric Systems and Applications. Springer, 2004.

[2] J. Rao and X. Su, "A Survey of Automated Web Service Composition Methods," in *International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[3] A. Strunk, "QoS-Aware Service Composition: A Survey," *IEEE European Conference on Web Services*, 2010.

[4] W. Jiang, S. Hu, Z. Huang, Z. Liu, and Q. D. Handler, "Two-Phase Graph Search Algorithm for QoS-Aware Automatic Service Composition," in *International Conference on Service-Oriented Computing and Applications*, 2010.

[5] Y. Yan, B. Xu, Z. Gu, and S. Luo, "A QoS-Driven Approach for Semantic Service Composition," in *IEEE International Conference on Commerce and Enterprise Computing*, 2009.

[6] M. Aiello, E. E. Khoury, A. Lazovik, and P. Ratelband, "Optimal QoS-Aware Web Service Composition," in *IEEE International Conference on Commerce and Enterprise Computing*, 2009.

[7] M. Chen and Y. Yan, "Redundant Service Removal in QoS-Aware Service Composition," *IEEE International Conference on Web Services (ICWS)*, 2012.

[8] Z. Zheng, H. Ma, M. R. Lyu, and I. King, "Collaborative web service qos prediction via neighborhood integrated matrix factorization," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 289–299, 2013.

[9] S. Dustdar and W. Schreiner, "A survey on web services composition," *International Journal of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.

[10] P. Bertoli, M. Pistore, and J. Hoffmann, "Web Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty," *Artificial Intelligence*, pp. 1013–1018, 2007.

[11] S. R. Ponnekanti and A. Fox, "SWORD : A Developer Toolkit for Web Service Composition," in *11th World Wide Web Conference*, 2002.

[12] M. Carman, L. Serafini, and P. Traverso, "Web Service Composition as Planning," in *Workshop on planning for web services (ICAPS 2003)*, 2003, pp. 1636—-1642.

[13] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN planning for Web Service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, 2004.

[14] E. Sirin and B. Parsia, "Planning for Semantic Web Services," in *Semantic Web Services Workshop at 3rd International Semantic Web Conference*, 2004, pp. 33–40.

[15] M. Klusch, A. Gerber, and M. Schmidt, "Semantic Web Service Composition Planning with OWLS-Xplan," in *Proceedings of the AAAI Fall Symposium on Semantic Web and Agents*, 2005.

[16] R. Akkiraju, B. Srivastava, A. A. Ivan, R. Goodwin, and T. Syeda-Mahmood, "SEMAPLAN: Combining planning with semantic matching to achieve Web service composition," in *IEEE International Conference on Web Services (ICWS 2006)*, 2006, pp. 37–44.

[17] O. Hatzi, D. Vrakas, M. Nikolaidou, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "An Integrated Approach to Automated Semantic Web Service Composition through Planning," *IEEE Transactions on Services Computing*, pp. 1–14, 2011.

[18] S. Oh, D. Lee, and S. R. T. Kumara, "Web service planner (WSPR): an effective and scalable web service composition algorithm," *International Journal of Web Service Research*, vol. 4, no. 1, pp. 1–22, 2007.

[19] 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*, vol. 30, no. 5, pp. 311–327, 2004.

[20] T. Yu and K.-J. Lin, "Service selection algorithms for Web services with end-to-end QoS constraints," *Information Systems and e-Business Management*, vol. 3, no. 2, pp. 103–126, 2005.

[21] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, "Heuristics for QoS-aware Web Service Composition," in *International Conference on Web Services (ICWS'06)*, 2006, pp. 72–82.

[22] M. Alrifai and T. Risse, "Combining global optimization with local selection for efficient QoS-aware service composition," in *18th International Conference on World Wide Web (WWW '09)*, 2009, pp. 881–890.

[23] G. Zou, Q. Lu, Y. Chen, R. Huang, Y. Xu, and Y. Xiang, "Qos-aware dynamic composition of web services using numerical temporal planning," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 18–31, 2014.

[24] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in *Genetic and Evolutionary Computation Conference, GECCO 2005*, 2005, pp. 1069–1075.

[25] H. Wada, J. Suzuki, Y. Yamano, and K. Oba, "E^3: A multiobjective optimization framework for sla-aware service composition," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 358–372, 2012.

[26] W. Jiang, S. Hu, and Z. Liu, "Top K Query for QoS-Aware Automatic Service Composition," *IEEE Transactions on Services Computing*, vol. 7, no. 4, pp. 681–695, Oct. 2014.

[27] L. Barakat, S. Miles, I. Poernomo, and M. Luck, "Efficient Multi-Granularity Service Composition," in *IEEE International Conference on Web Services (ICWS)*, 2011.

[28] F. Wagner, F. Ishikawa, and S. Honiden, "QoS-aware Automatic Service Composition by Applying Functional Clustering," in *IEEE International Conference on Web Services (ICWS)*, 2011, pp. 89–96.

[29] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic web service composition with a heuristic-based search algorithm," in *IEEE International Conference on Web Services (ICWS 2011)*, 2011, pp. 81–88.

[30] P. Rodriguez Mier, C. Pedrinaci, M. Lama, and M. Mucientes, "An Integrated Semantic Web Service Discovery and Composition Framework," *IEEE Transactions on Services Computing*, 2015 (DOI 10.1109/TSC.2015.2402679).

[31] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic Matching of Web Services Capabilities," in *The Semantic Web - ISWC 2002*, 2002, pp. 333–347.

[32] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue, "iServe: a linked services publishing platform," in *CEUR Workshop Proceedings*, vol. 596, 2010.

[33] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut, "Quality of service for workflows and web service processes," *Web Semantics*, vol. 1, pp. 281–308, 2004.

[34] B. Carré, *Graphs and networks*. Oxford University Press, 1979.

[35] P. Rodríguez-Mier, M. Mucientes, J. C. Vidal, and M. Lama, "An Optimal and Complete Algorithm for Automatic Web Service Composition," *International Journal of Web Service Research*, vol. 9, no. 2, pp. 1–20, 2012.

[36] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "A Dynamic QoS-Aware Semantic Web Service Composition Algorithm," in *International Conference on Service-Oriented Computing*, 2012.

[37] J. a. L. Sobrinho, "Algebra and algorithms for QoS path computation and hop-by-hop routing in the Internet," in *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, vol. 10, no. 4, 2002, pp. 541–550.

[38] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[39] S. Kona, A. Bansal, M. B. Blake, S. Bleul, and T. Weise, "WSC-2009: a quality of service-oriented web services challenge," in *IEEE International Conference on Commerce and Enterprise Computing*, 2009.

[40] T. Weise, "Web Service Challenge 2010," http://www.it-weise.de/documents/files/W2010WSC_pres.pdf, 2010, [Online; accessed May-2015].

[41] I. Rodríguez-Fdez, A. Canosa, M. Mucientes, and A. Bugarín, "STAC: a web platform for the comparison of algorithms using statistical tests," in *Proceedings of the 2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2015.