

# An Integrated Semantic Web Service Discovery and Composition Framework

Pablo Rodriguez-Mier, Carlos Pedrinaci, Manuel Lama, and Manuel Mucientes

**Abstract**—In this paper we present a theoretical analysis of graph-based service composition in terms of its dependency with service discovery. Driven by this analysis we define a composition framework by means of integration with fine-grained I/O service discovery that enables the generation of a graph-based composition which contains the set of services that are semantically relevant for an input-output request. The proposed framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimisations to improve the scalability of the system. A practical implementation used for the empirical analysis is also provided. This analysis proves the scalability and flexibility of our proposal and provides insights on how integrated composition systems can be designed in order to achieve good performance in real scenarios for the Web.

**Index Terms**—Semantic Web Services; Service Discovery; Service Composition Framework; Service Composition Performance.

## 1 INTRODUCTION

SERVICE discovery and composition are in general complex tasks that require considerable effort, especially when vast amounts of services are available. Service discovery solutions range from the initial UDDI proposal that relied on the syntactic description of services and a prefixed categorisation [1], to more advanced generic solutions able to discover Web APIs and Web services across domains exploiting rich user-provided semantic service descriptions [2]. Similarly, a plethora of service composition solutions have been produced spanning from mere graphical support to completely automated solutions [3]–[5]. Both discovery and composition engines essentially rely on the processing of service descriptions, which increasingly go beyond syntactic representations to include the semantics of the service(s) to enable more advanced computations [6], [7].

An analysis of the service composition literature highlights that, regardless of the approach, a central task that needs to be frequently performed throughout the composition activity, is the discovery of suitable services to use. Whether one looks at fully automated composition engines based on Artificial Intelligence (AI) planning techniques [8]–[10], or at more constrained solutions that rely on pre-defined skeletal plans [11], [12], or at graph based approaches focused on semantic input-output parameter matching [13]–[20], service discovery

is a central activity that needs to be carried out at every main step during the generation of the composition. Yet, despite the strong dependency between both activities, research and development in both areas has evolved for the most part independently.

On the one hand, service discovery has traditionally been approached as a one-of activity to be sporadically carried out by humans when looking for services. As a consequence the interface exposed by discovery engines assumes that requests are fully specified in terms of a well-defined interface and categorisation. Moreover, response times of discovery engines are orders of magnitude above what would be acceptable for a composition engine that should it delegate the thousands discovery requests it needs to issue at composition time [21]. These limitations hamper the development of fast composition systems where discovery and composition are two fundamental, interrelated activities.

On the other hand, partly due to the particularly demanding computational needs of composition algorithms, most composition engines reimplement locally their own discovery methods instead of integrating existing components providing state of the art discovery algorithms. Additionally, this approach relies on the unnecessary and often unrealistic assumption that the entire set of services should be locally available to the composition engine. This assumption requires pre-importing all services locally which is only viable for those registries providing entire public dumps of the service descriptions they hold. Furthermore, most composition engines do not introduce optimisation techniques to improve the scalability by identifying equivalent or dominant functionality that could appear when many different service registries are involved in the composition. This prevents the use of optimal search strategies since the complexity usually grows exponentially with the number of services.

- P. Rodriguez-Mier, M. Lama and M. Mucientes work at the Centro de Investigación en Tecnologías da Información (CITIUS), Universidade de Santiago de Compostela, Spain.  
E-mail: {pablo.rodriguez.mier,manuel.lama,manuel.mucientes}@usc.es
- Carlos Pedrinaci is with The Open University, Milton Keynes, UK.  
E-mail: carlos.pedrinaci@open.ac.uk

© 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.

In order to tackle the previous problems, a composition framework should consider the following characteristics: 1) provide convenient *fine-grained discovery* mechanisms that could help to discover services able to consume or produce (a subset of) certain types of data as usually required during composition; 2) improve the *response time* of service discovery to process requests very fast; 3) support the integration of *third party service registries* as a key activity in the composition phase; 4) incorporate *optimizations* to improve the scalability of the overall composition process; and 5) find *optimal service compositions* by minimizing different criteria such as the number of services or the length of the composition to avoid complex, unmanageable solutions.

In this paper we present a graph-based framework focused on the semantic input-output parameter matching of services' interfaces that efficiently integrates the automatic service composition and semantic service discovery. The provided framework takes into account all the characteristics indicated in the above paragraph. Notably, the main contributions are:

- 1) A formal framework that presents a theoretical analysis of graph-based service composition in terms of its dependency with a service discovery and we provide a fine-grained I/O discovery interface which reduces the performance overhead without having to assume the local availability and in-memory preloading of service registries. The framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimisations to improve the scalability of the system, which as far as we now are not included in other frameworks.
- 2) A reference implementation of this formal framework based on the adaptation of two independently developed components, namely ComposIT [22] and iServe [2], respectively in charge of service composition and discovery.
- 3) A detailed performance analysis of the integrated system, highlighting both the unacceptable performance achieved when using the typical out of the box discovery implementations, as well as the fact that top performance is achievable with the adequate discovery granularity and corresponding indexing optimisations.

The proposed framework is data-flow centric, focused on the semantic I/O parameter matching of services' interfaces and leaving aside preconditions and effects. This is essentially a pragmatic decision inline with the current tendency towards lightweight data-driven approaches. In fact, on the Web less than 5% of the semantic Web services include preconditions and effects [23].

The rest of the paper is organized as follows. Sec. 2 discusses the state-of-the-art. Sec. 3 formalizes the web service composition problem and Sec. 4 framework that defines the composition in terms of service discovery

tasks. Sec. 5 describes our reference implementation. Sec. 6 explores the performance of the system for different scenarios and finally Sec. 7 gives some final remarks.

## 2 RELATED WORK

Automatic composition of Web services is still an open problem that involves multiple research areas [5]. Concretely, lots of efforts have been devoted to automate the discovery and composition using different approaches and techniques [24]. However, most of the research in both areas has been evolved independently of each other, despite the significant overlap between these interrelated tasks. This has led to a lack of integrated approaches in the field that consider the performance and the scalability of the overall integrated system as well as the impact of the discovery in terms of response time during the automatic composition task.

From the discovery side, most of the work has been focused on improving the retrieval performance (i.e., precision-recall curve) without much concern about the response time requirements and/or the interface requirements to provide an efficient fine-grained discovery granularity for automatic composition. However, the response time of the discovery systems is recently gaining significant interest. A recent service discovery competition [21] shows some of the newest advances in the automatic discovery field. Most relevant examples are OWLS-MX3 [25], iSem 1.1 [26] and XSSD [27]. The main conclusions that can be drawn from this contest, from the perspective of service composition, are twofold: 1) research efforts are focused on response time improvement via caching and indexing, yet still not sufficient for fast, automatic composition of services and 2) the interface exposed by discovery engines assumes that requests are fully specified in terms of a well-defined interface and categorisation, i.e., discovery systems expect a precise description of the service in terms of inputs and outputs, and/or other characteristics such as preconditions and effects. However, these interfaces are not adequate for service composition, since one of the assumptions is that there is usually no single service that fully matches a request and therefore several services need to be combined instead. Indeed, during automatic composition, an exploratory search is usually required to guess which relevant services can be selected at each step. This requires to launch many partial requests (fine-grained queries), rather than fully specified requests, in order to locate relevant services that match some partial information available to the algorithm (e.g., services that consume some inputs and/or produce some outputs). Fine-grained requests are simpler and can be solved faster than complex, fully specified requests. Thus they are more suitable for automatic composition systems.

From the composition side, most approaches can be categorized into: 1) classical AI planning approaches [28], where the composition problem is translated into the planning domain and solved using general planners,

and 2) graph-based I/O driven approaches that build a graph with the services and their input/output semantic relations (generally omitting the preconditions and effects), and apply graph search techniques to extract (usually optimal) service compositions from the graph.

Relevant approaches of the first group are [9], [10], [29]. These approaches differ from our work in the sense that they handle very expressive preconditions and effects to generate composition plans but: 1) the concept of external service registries is missing, services are assumed to be locally available; 2) average response time of these systems is usually high; and 3) optimizations to reduce the number of services by identifying redundant functionality are not considered.

On the other hand, graph-based I/O approaches are gaining much attention since the Web Service Challenge [30]. Some notable works in this field are [14]–[20]. Concretely, [14], [15], [20] are the top-3 algorithms of the WSC'08. Although these approaches show generally good performance and low response times, [14] and [15] do not find optimal solutions and [20] fails to find solutions in large data sets. Additionally, none of these systems consider neither the integration with service registries nor the use of service optimizations to deal with potential scalability problems.

From the point of view of the integrated frameworks, a very interesting approach was proposed by Kona et al. in [13]. In this paper, the authors present an efficient framework for Web service composition that supports semantic Web service discovery. The composition is generated by performing a forward chaining of operators to find a feasible composition. The authors also evaluated the system with the datasets of the Web Service Challenge 2006 and presented a detailed experimentation. Their results demonstrate the capabilities and the good performance of this system which, however, exhibits some limitations: 1) the notion of an external service registry is missing, all the information required is pre-processed and loaded in the main memory, which is one of the main issues we set out to tackle with this work since it is otherwise not possible to deal with large and/or distributed datasets; 2) the framework does not contemplate service optimizations to remove redundant information and 3) the work does not perform an optimal search to minimise the cost or the number of services of the composition as all possible compositions with the shortest length are captured in the composition graph which should be further processed to extract the optimal composition. Similarly, in [31], Lécué et al. develop an integrated framework for dynamic Web service composition. The framework exploits the semantic input-output matchmaking to discover relevant services and performs automatic composition using a graph-based approach, taking into account functional and non-functional properties. However, graph optimizations are not considered and the composition search is non-optimal, since the selection of the services is merely greedy-based.

In [32], Da Silva et al. present a framework that

effectively supports both automatic semantic discovery and composition, among other relevant phases of the composition life-cycle, such as service publication and service selection, taking into account non-functional properties. One of the limitations of the discovery phase is that it does not support fine-grained requests. On the other hand, the framework does not include neither optimizations to reduce graph size nor an optimal search to extract the best composition from the graph.

In light of the above analysis, we propose a graph-based I/O framework that overcomes all of the analyzed limitations. In this framework the discovery is defined in terms of a fine-grained I/O interface which minimises the performance overhead between both composition and discovery without having to assume the local availability and in-memory preloading of service registries. The proposed framework also includes an optimal composition search algorithm to extract the best composition from the graph minimising the length and the number of services, and different graph optimizations to improve the scalability of the system.

### 3 WEB SERVICE COMPOSITION PROBLEM

Service composition aims to help construct composite services that could fulfil a user request, e.g., booking an entire holiday, when no known service can achieve such a request on its own. A core activity for creating service compositions is, indeed, the discovery of relevant services. In this context, relevant services are those that could be invoked and contribute to obtaining an executable composition that would fulfil the needs set out by the client. We herein formalise the composition problem in close relationship with discovery as a means to better study and approach the integration of discovery and composition engines. The formalisation of the problem is data-flow centric, focussed on the semantic input-output parameter matching of services' interfaces.

#### 3.1 Semantic Web Service Discovery

The semantic Web service discovery problem consists of locating appropriate services from one or more service registries that are relevant to an input-output request.

*Definition 1:* A Semantic Web Service (SWS, hereafter "service") can be defined as a tuple  $w = \{In_w, Out_w\} \in W$  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, and  $W$  is the set of all services available in the service registry. Each input and output is related to a semantic concept from an ontology  $O$  ( $In_w, Out_w \subseteq O$ ).

Semantic inputs and outputs can be used to discover relevant services as well as 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 typically contemplated in the literature are [33]:

- **Exact** ( $\equiv$ ): An output  $o_{w1} \in Out_{w1}$  of a service  $w1$  matches an input  $i_{w2} \in In_{w2}$  of a service  $w2$  with a degree of exact match if both concepts are equivalent.
- **Plugin** ( $\sqsubseteq$ ): An output  $o_{w1} \in Out_{w1}$  of a service  $w1$  matches an input  $i_{w2} \in In_{w2}$  of a service  $w2$  with a degree of plugin if  $o_{w1}$  is a sub-concept of  $i_{w2}$  ( $o_{w1} \sqsubseteq i_{w2}$ ).
- **Subsume** ( $\supseteq$ ): An output  $o_{w1} \in Out_{w1}$  of a service  $w1$  matches an input  $i_{w2} \in In_{w2}$  of a service  $w2$  with a degree of subsume if  $o_{w1}$  is a super-concept of  $i_{w2}$  ( $o_{w1} \supseteq i_{w2}$ ).
- **Fail** ( $\perp$ ): When none of the previous matches are found, then both concepts are incompatible and the match has a degree of fail ( $o_{w1} \perp i_{w2}$ ).

Note that, in order to discover relevant services to generate data-flow compatible service compositions, the only two valid degrees of match are *exact* and *plugin*. On this basis, we define the *cmatch* (compatible match) function that will be used to discover candidate services during the composition phase:

*Definition 2:* Given  $a, b \in O$ , a compatible match  $cmatch(a, b)$  holds if and only if  $a \equiv b$  (exact match) or  $a \sqsubseteq b$  (plug-in match).

Using the previous compatible match function between concepts, we can define the matchmaking operator " $\otimes$ " that given two sets of concepts  $C_1, C_2 \subseteq O$ , it returns the concepts from  $C_2$  matched by  $C_1$ .

*Definition 3:* Given  $C_1, C_2 \subseteq O$ , we define " $\otimes : O \times O \rightarrow O$ " such that  $C_1 \otimes C_2 = \{c_2 \in C_2 | cmatch(c_1, c_2), c_1 \in C_1\}$ . Note that this operator is not commutative.

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

*Definition 4:* Given  $C_1, C_2 \subseteq O$ , a full matching between  $C_1$  and  $C_2$  exists if  $C_1 \otimes C_2 = C_2$ , whereas a partial matching exists if  $C_1 \otimes C_2 \subset C_2$ .

Typically, a service  $w = \{In_w, Out_w\}$  is relevant to a request  $r = \{In_r, Out_r\}$ , where  $In_r \subseteq O$  are the provided inputs and  $Out_r \subseteq O$  the expected outputs, if  $In_r \otimes In_w = In_w$  and  $Out_w \otimes Out_r = Out_r$ , that is, there is a full match between the provided inputs and the service inputs and a full match between the service outputs and the expected outputs.

While this approach is reasonable for discovering the services that best match an entire request (full match), for composition one needs to locate services that are relevant, that is, that match some inputs / outputs (partial match). Thus, rather than approaching the discovery problem based on a full input/output description, we split this problem into two finer-grained discovery problems that are more relevant for service composition: input discovery and output discovery.

*Definition 5:* Given a set of concepts  $C \subseteq O$ , the input discovery problem can be defined as finding a set of relevant services  $W = \{w_1, \dots, w_n\}$  where  $w_i = \{In_{w_i}, Out_{w_i}\}$  such that  $\forall w_i \in W, C \otimes In_{w_i} \subseteq In_{w_i}$ , that is, services that can consume some (partial match) of the inputs or are directly invocable (full match) with  $C$ .

*Definition 6:* Given a set of concepts  $C \subseteq O$ , the output discovery problem can be defined as finding a set of relevant services  $W = \{w_1, \dots, w_n\}$  where  $w_i = \{In_{w_i}, Out_{w_i}\}$  such that  $\forall w_i \in W, Out_{w_i} \otimes C \subseteq C$ , that is, services that produce some or all outputs.

Based on these definitions, we introduce the notion of input and output relevance:

*Definition 7:* A service  $w = \{In_w, Out_w\}$ , where  $In_w, Out_w \subseteq O$ , is *input-relevant* for a set of concepts  $C \subseteq O$  if  $C \otimes In_w \neq \emptyset$ , whereas the service  $w$ , is *output-relevant* for a set of concepts  $C \subseteq O$  if  $Out_w \otimes C \neq \emptyset$ .

### 3.2 Semantic Web Service Composition

The semantic composition problem considered in this work is as follows: Given a request  $r = \{In_r, Out_r\}$ , where  $In_r$  is a set of available semantic input concepts and  $Out_r$  a set of requested semantic output concepts, we can define the problem of the automatic construction of a SWS composition as that of finding a composite Web service  $w_c = \{In_{w_c}, Out_{w_c}, P = \{S, \leq\}\}$  such that  $In_r \otimes In_{w_c} = In_{w_c}$  (the composite service is invocable with the available inputs) and  $Out_{w_c} \otimes Out_r = Out_r$  (the composite service retrieves all the requested outputs). This service consists of a partially ordered set  $P$  (a binary relation " $\leq$ " over a set of services  $S \subseteq W$ ). This partial ordered set of services is essentially a Directed Acyclic Graph (DAG) which models the implicit execution order of the services driven by the input/output matches, where nodes of the DAG are services and the arcs are valid semantic matches. This type of composition has many advantages: On one hand, mapping inputs and outputs to semantic concepts does allow to reason about data types to improve the matchmaking between service parameters, which leads to more possible semantically valid compositions. On the other hand, DAG representation formally captures the nature of a composition where services may be executed in different orders, i.e., there are many different total (sequential) orderings of a composition that lead to the same result. Moreover, since our approach is data-flow centric, a DAG representation is simpler than a general (possible cyclic) graph as cycles do not produce new data types in the composition.

However there are also some drawbacks. First, a DAG representation could impose some restrictions in the compositions that can be generated, i.e, due the absence of cycles, a service could not explicitly be invoked twice. Second, compositions at different semantic levels rather than just concept matchmaking would definitely improve the quality of the compositions by capturing more possible cases. Furthermore, using input concepts and output concepts to define a composition request is not user friendly. A better way to specify a request would be to define it with keywords. This, nonetheless, could be achieved with a pre-processing step using automatic semantic annotation tools to translate the request from keywords to semantic concepts. Formally, we define a valid composition as follows:

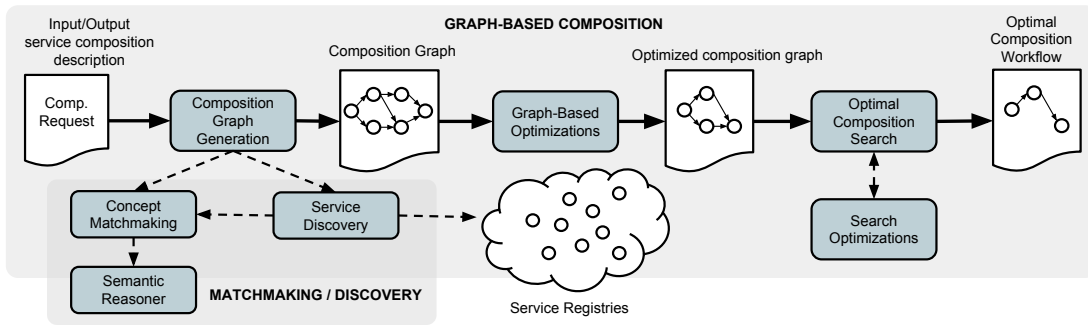


Fig. 1. Overview of the proposed approach.

*Definition 8:* Let  $r = \{In_r, Out_r\}$  and let  $w_c = \{In_{w_c}, Out_{w_c}, P = \{S, \leq\}\}$  be a composite service for the request  $r$ , where  $P$  is a partial order over the set of services  $S \subseteq W$  of the composite service  $w_c$ . We say  $w_c$  is a valid composition for request  $r$  if and only if, for any topological sort  $T = \{w_1, w_2, \dots, w_N\}$  of  $P$ , where  $w_j = \{In_{w_j}, Out_{w_j}\} \forall j \in [1, N]$ , the following expression is satisfied:

$$(In_r \otimes In_{w_1} = In_{w_1}) \wedge ((In_r \cup Out_{w_1}) \otimes In_{w_2} = In_{w_2}) \\ \wedge \dots \wedge ((In_r \cup Out_{w_1} \cup \dots \cup Out_{w_N}) \otimes Out_r = Out_r).$$

This definition implies that every service of the composition must be invocable to obtain an invocable service composition. We say that a service  $w = \{In_w, Out_w\}$  is invocable with a set of concepts  $C \subseteq O$  if each required input  $i_w \in In_w$  is semantically matched by a set of concepts  $C$ .

*Definition 9:* If  $C \subseteq O$  is the set of available input concepts, then a service  $w = \{In_w, Out_w\}$  is *invokable* with  $C$  if  $C \otimes In_w = In_w$ , i.e., there exists a full matching between the available inputs and the service inputs.

Note that if a service  $w$  is invocable with a set of concepts  $C$ , then it is also input-relevant for the same set of concepts since *invokable* implies *input-relevant*, but the inverse does not hold (see Def. 7). That is, the set of invocable services is included in the set of the relevant services.

The reader should note that we restrict the definition of a compatible match to *exact* and *plugin* in order to generate semantically complete compositions. However, the framework also supports the use of other match degrees (e.g., *subsume*) by relaxing the “*ematch*” operator, which in practice means obtaining potentially more matched (but semantically weaker) concepts and thus bigger composition graphs with more services and match relations that could be semantically incomplete. This is supported not only in theory, but also by the reference implementation presented in Sec. 5.

## 4 COMPOSITION FRAMEWORK

On the basis of the formal definition of the problem, in this section we present a graph-based framework for automatic semantic Web service composition. Fig. 1

shows the overview of our approach with the different steps involved. The process is triggered by a composition request that specifies the user requirements in terms of inputs and the expected outputs. This information is used in the composition graph generation phase to build a graph with all the relevant services and the semantic relations between their inputs and outputs. In order to find the relevant services, the composition graph phase is interleaved with the *discovery phase*. The discovery phase is responsible for retrieving the relevant services given the data available at different stages during the *composition graph generation phase*. The relationships between the inputs and outputs of services are computed in the *matchmaking phase*, where the semantic matching degree between inputs and outputs is computed using a semantic reasoner. The service composition graph is eventually generated on the basis of the relevant services and the I/O matching information. This graph contains all possible service compositions that satisfy the composition request, in addition to a few others that, although invocable, do not manage to entirely fulfil the request. The service composition graph is then optimised applying different techniques to group and reduce the number of services and relations. Next, an *optimal search* is performed over the graph to find the optimal composition. This phase is interleaved with a *search optimisation phase* that analyses and reduces the search space. Finally, the optimised composition workflow is returned.

In this section, we analyse each phase and we provide generic strategies based on the problem description presented in the previous section.

### 4.1 Semantic Matchmaking

A fundamental functionality that needs to be available for generating compositions and even for discovering services, is the ability to analyse the compatibility between different semantic types. This functionality, which we refer to as semantic matchmaking, is in charge of assessing the level of semantic compatibility between concepts, given an ontology (or set of ontologies). To do so, semantic matchmaking relies on semantic reasoning (notably subsumption reasoning) in order to be able to determine the relationships between the concepts (e.g., *Plugin match*). This mechanism can be used for example,

to discover services that can consume or produce a concrete input/output by finding semantically compatible types. Such a mechanism is also particularly relevant for generating the service composition graph with all the matches between services inputs and outputs.

The matchmaking system provides a  $match(C_1, C_2)$  function which represents the concrete implementation function of the  $\otimes$  operator defined in Def. 3. The  $match$  function tries to find a valid match between the *source concepts* of  $C_1$  and the *target concepts* of  $C_2$  calling the  $cmatch(c_i, c_j)$  function (Def. 2) for each pair  $(c_i, c_j)$  of concepts where  $c_i \in C_1$  and  $c_j \in C_2$ . The compatible match function is calculated using a semantic reasoner that returns the semantic relation between two concepts. Then, it checks if the relation is considered a compatible match (i.e., *exact* or *plugin*). Each time a compatible match is found between  $c_i$  and  $c_j$ ,  $c_j$  is added to a set of matched concepts and removed from  $C_2$ . The reader should note that the goal here is not to find the best match for each element but rather to get all compatible matches for each target element.

The best-case complexity (all  $C_2$  concepts matched by the first element from  $C_1$ ) is  $O(m)$ , whereas the worst-case complexity (no compatible matches at all) is  $O(m \cdot n)$  where  $n = |C_1|, m = |C_2|$ . This implies that, in the worst case, for two sets of elements, there will be at most  $m \times n$  calls to the  $cmatch$  function which is ultimately answered by the semantic reasoner.

## 4.2 Semantic Service Discovery

In order to generate service compositions, it is necessary to be able to discover appropriate services based on their interface. The goal of a typical discovery system is to find atomic services that match entirely a description representing the ideal service sought, i.e., all the inputs and outputs are compatible. However, from the viewpoint of generating data-flow compatible compositions, rather than looking for entire matches, we need to find suitable combinations of services that combined would satisfy a request. In this scenario, the ability to find partially matching services very fast is paramount in order to enable exploring efficiently the many possible combinations of services that could lead to a suitable composition. Therefore, in a nutshell, the type of service discovery that is required for supporting service composition is a more relaxed and finer-grain version of that typically provided by discovery engines whereby partial matches can be obtained in a very fast manner. This can be achieved by defining a simple fine-grained interface that supports the discovery of services using only partial information (some/any available inputs, some/any expected outputs). Fig. 2 shows the pseudocode of this simple interface to discover relevant services that can be used as a starting point to obtain semantic input/output relevant services, as defined in Def. 7 in Sec. 3.

The discovery algorithm sequentially scans all services and calls the  $Match$  function of the  $Matchmaker$  to determine if a service is relevant for an input (the service has

```

1: function RELEVANTIO( $C \subseteq O, W, type$ )
2:    $relevantServ := \{\}$ 
3:   for all  $w_i = \{I_{w_i}, O_{w_i}\} \in W$  do
4:     if  $type = In$  then
5:       if  $match(C, I_{w_i})$  then
6:          $relevantServ := relevantServ \cup w_i$ 
7:       end if
8:     else if  $type = Out$  then
9:       if  $match(O_{w_i}, C)$  then
10:         $relevantServ := relevantServ \cup w_i$ 
11:      end if
12:    end if
13:  end for
14:  return  $relevantServ$ 
15: end function

```

Fig. 2. Pseudocode to obtain input-relevant and output-relevant sets of services for a particular set of concepts

at least one input compatible with the inputs provided) or for an output (the service has at least one output compatible with the outputs provided) depending on the  $Type$  selected. Therefore, the complexity of this type of discovery is  $O(w)$  where  $w = |W|$  is the size of the service repository. This implies at most  $|W|$  calls to  $Match$  in the worst-case scenario or  $O(w \cdot m \cdot n)$  if we consider the complexity of the  $Match$  method assuming every service has at most  $m$  outputs and  $n$  inputs.

## 4.3 Service Composition Graph Generation

When the system receives a request, the *Service Composition Graph Generator* computes a graph with all the semantic relations between the relevant services for the request. A request is basically a set of input concepts, which represent the initial set of available inputs, and a set of output concepts, which are the outputs that the composite service should return. The *service composition graph* is basically a layered Directed Acyclic Graph (DAG),  $G = (V, E)$ , where:

- $V = W \cup C$  is the set of vertices of the graph, where  $W$  is the set of services and  $C$  the set of concepts (inputs and outputs).
- $E = CW \cup WC \cup CC$  is the set of edges in the graph where:
  - $CW \subseteq \{(c, w) \mid c, w \in V \wedge c \in C \wedge w \in W\}$  is the set of input edges, i.e., edges connecting input concepts to their services.
  - $WC \subseteq \{(w, c) \mid w, c \in V \wedge w \in W \wedge c \in C\}$  is the set of output edges, i.e., edges connecting services with their output concepts.
  - $CC \subseteq \{(c, c') \mid c, c' \in V \wedge c, c' \in C \wedge cmatch(c, c')\}$  is the set of edges that represent a semantic match between concepts.

This graph contains all the known services that could directly or indirectly be invoked given the provided inputs. The graph is divided into  $N$  layers, whereby each layer  $i$  has all those services whose inputs are matched by the outputs produced in previous layers and, therefore, are invocable at layer  $i$ . The graph is augmented with two layers, namely  $L_0$  and  $L_{N+1}$ .  $L_0$

contains the dummy service  $w_O = \{O_R, \emptyset\}$  whereas  $L_{N+1}$  contains the dummy service  $w_I = \{\emptyset, I_R\}$ . The first one is a service that provides as outputs the inputs of the request ( $I_R$ ) and the last one has the goal outputs ( $O_R$ ) as inputs. An example of a graph for  $I_R = \{BookTitle, BookAuthor, CreditCard, Email, Address\}$  and  $O_R = \{Price, Payment, BookingCode\}$  is shown in Fig. 3.

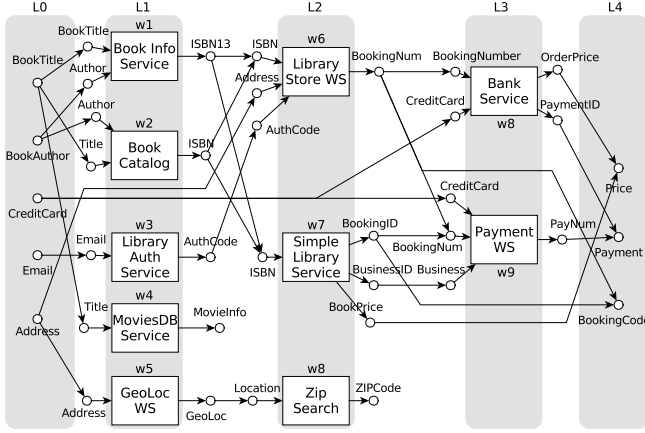


Fig. 3. Composition graph example.

The first step of the composition graph construction is the calculation of the relevant services. These services can be easily calculated forwards, layer by layer, using the discovery mechanism previously presented. Fig. 4 shows an implementation of the forward composition graph generation algorithm for a request  $R$ . The algorithm selects all those services from the set of all available services  $W$  that are input-relevant for the available concepts ( $availCon$ ) in each layer using the *relevantIO* function (L. 8). Then, for each *input-relevant* service, the algorithm performs a match between the available concepts and the unmatched inputs of each service. All the inputs that are matched are removed from the unmatched set of inputs for the current service. If there are no unmatched inputs, then the service is invocable and thus is eligible for the current 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 concepts 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. Note that instead of performing the invocability check by finding a full match between  $C$  and the inputs of each service, we save those inputs of each service that have been matched before, and hence we only perform the match between the new outputs generated in the previous level ( $availCon$ ) and the remaining unmatched inputs of each service ( $U_{set}$ ). Hence, the unmatched inputs  $U_{set}$  of each service decreases monotonically with each level (i.e., the unmatched inputs of each service always decrease when a new match is found, and the effect is propagated at each layer). The complexity analysis for this algorithm (neglecting the optimisation

```

1: function FWDGRAPH( $R = \{I_R, O_R\}, W$ )
2:    $C := I_R$ ;  $i := 0$ ;  $L_0 := \{w_I\}$ ;  $L := L_0$ 
3:    $unmatchedIn := []$ ;  $availCon := I_R$ 
4:    $W' := W$ ;
5:   repeat
6:      $i := i + 1$ 
7:      $L_i := \emptyset$ ;  $W_{selected} = \emptyset$ 
8:      $W_{relevant} := relevantIO(availCon, W', In)$ 
9:      $availCon := \emptyset$ 
10:    for all  $w_i = \{I_{w_i}, O_{w_i}\} \in W_{relevant}$  do
11:       $U_{set} := unmatchedIn[w_i]$ 
12:       $M_{set} := Match(availCon, U_{set})$ 
13:       $unmatchedIn[w_i] := U_{set} \setminus M_{set}$ 
14:      if  $M_{set} = \emptyset \wedge w_i \notin L$  then
15:         $W_{selected} = W_{selected} \cup w_i$ 
16:         $availCon := availCon \cup O_{w_i}$ 
17:      end if
18:    end for
19:     $L_i := L_i \cup W_{selected}$ 
20:     $W' := W' \setminus W_{selected}$ 
21:     $C := C \cup availCon$ 
22:  until ( $Match(C, O_R) = O_R$ )  $\vee L_i = \emptyset$ 
23:   $L := L \cup \{w_O\}$ 
24: end function

```

Fig. 4. Algorithm for forward graph generation.

effect due to the propagation of the matched inputs for simplification purposes) is  $O(l \cdot w \cdot m \cdot n + l \cdot \frac{w}{k} \cdot m \cdot n)$  which can be simplified to  $O(l \cdot m \cdot n \cdot (\frac{(k+1)w}{k}))$ . The first part corresponds with the complexity of the calls to the *relevantIO* function which is invoked  $l$  times (one call per layer), whereas the second part corresponds with the complexity of the *for* loop to check the invocability of each *input-relevant* service. We can expect that only a small subset of the repository  $W$  is relevant for the *availCon* generated in the previous layer. Thus, each call to *relevantIO* function returns a small set of relevant services  $w/k$  where  $k$  ( $k \gg 1$ ) is a reduction factor that depends on the number of relevant services for a given set of concepts. This  $k$  factor is different for each request and service registry. For example, if we assume  $k = 100$  for a given problem for a service registry of 1,000 services, then it means that each invocation of *relevantIO*( $availCon, W, In$ ) will return only the 1% of the services of the repository ( $w/k = 10$ ). Consider the following example of a composition over a repository with 1,000 services ( $w = 1,000$ ), assuming that there are  $m = 5$  new output concepts generated and  $n = 5$  unmatched concepts at each layer, the composition graph has 10 layers ( $l = 10$ ) and in each layer the *relevantIO* function returns on average  $w/k = 10$  services (that is,  $k = 100$ ). The complexity in this example is  $10 \cdot 1000 \cdot 5 \cdot 5$  for the first part plus  $10 \cdot \frac{1000}{100} \cdot 5 \cdot 5$  for the second part, which is  $\approx 2.5 \cdot 10^5$  calls to the matchmaking system to compute all the required matches at the concept level.

#### 4.3.1 Index-Based Optimisations

Although these improvements can save search time, one of the bottlenecks of the graph generation is still the size of the repository  $w$ , which is usually some orders of magnitude bigger than the other parameters involved in the complexity. One effective way to reduce

the impact of the size of the repository is precalculating and indexing the input-relevant set of services for each concept of the ontology. The indexing of services can be done independently of any composition request as it only depends on the information available, such as the services themselves and the ontologies.

The construction of an inverted index function to recover input-relevant services or output-relevant services can be done easily using the *relevantIO* function. The main idea behind the inverted index is to build a key-value hash map where the keys are the concepts of the ontology and the values are those services that are input-relevant (or output-relevant) for that concept. This map allows to discovery relevant services in constant time during the graph generation.

We define a new function *relevantIO'* which is the cached-version of the original function. Instead of computing the relevance by using directly the matchmaking system, it first checks if the concept is cached in the inverted index. If the concept is in the index, then it is immediately returned (constant time). If not, the call is delegated to the *relevantIO* function. Assuming there is enough memory to keep the entire index, the index allows to provide relevant services at  $O(1)$  for each concept during the forward graph generation. Thus, we reduce the complexity associated to the parameter  $w$ . Concretely, since we can obtain at constant time the input-relevant services for each concept, the complexity of *relevantIO*(*availCon*,  $W$ , *In*) now depends only on the number of concepts in *availCon* (one access to the index per concept). Having  $m = |\text{availCon}|$  (number of new concepts at each layer) the complexity using indexes is  $O(l \cdot m + l \cdot \frac{w}{k} \cdot m \cdot n)$ , simplified to  $O(l \cdot m(1 + \frac{w}{k} \cdot n))$ . The use of indexes to discover relevant services during the forward graph generation has a high impact on the global performance. Using the same example as before, with  $w = 1000$ ,  $l = 10$ ,  $m = 5$ ,  $n = 5$  and  $k = 100$  we have  $10 \cdot 5(1 + \frac{1000}{100} \cdot 5) = 2.55 \cdot 10^3$ , 2 orders of magnitude lower than the non-indexed version.

#### 4.4 Graph-Based Optimisations

Once the graph is generated, the next step is to apply different optimisations to reduce the graph size in order to improve the optimal composition search performance. This part of the composition is independent of the discovery phase. All the information required to search for the optimal composition is in the graph, namely, the relevant services and the semantic relations between their inputs and outputs, so there is no need to communicate with the discovery/matchmaking systems. We distinguish at least two different techniques [22], [34]: *backward pruning* and *interface dominance*.

##### 4.4.1 Backward pruning

As explained earlier, the generation of the composition graph with the relevant services is done forwards, layer by layer. During this forward expansion of the graph, we

are not interested in invoking services that have no explicit effects on the composition, that is, services that are not contributing to the output goals. When the graph is completed and the goal outputs are reached, a backward pruning is performed to remove all non-contributing services. A non-contributing service is essentially a service that is not contained in the *transitive closure set* of the *output-relevant* services. A service  $w' = \{In_{w'}, Out_{w'}\}$  is *output-relevant* for a service  $w = \{In_w, Out_w\}$  if  $Out_{w'} \otimes In_w \neq \emptyset$  (def. 7). Thus, the set of all *output-relevant* services for a service  $w$  can be defined as:

$$X(w) = \{w' \in W \mid Out_{w'} \otimes In_w \neq \emptyset\} \quad (1)$$

Recursively, we can define the set of  $X^2(w) = X(X(w))$  as the set of *output-relevant* services at the distance two. Extending this, the *transitive closure* of the *output-relevant services* can be defined as:

$$\hat{X}(w) = X(w) \cup X^2(w) \cup X^3(w) \cup \dots \quad (2)$$

Therefore, we can say that all those services of the graph that are not in the *transitive closure* of the *output-relevant* services  $\hat{X}$  are not contributing to the composition goals, directly nor indirectly, and can therefore be removed from the graph.

An example of this can be seen in Fig. 3. Starting from the last layer, we compute the transitive closure of the service  $w_O$ , which is a dummy service that represents the goal outputs. The output relevant services for  $w_O$  at distance one are  $X(w_O) = \{w_6, w_7, w_8, w_9\}$ , since  $Out_{w_O} \otimes In_{w_O} \neq \emptyset$  and the same for  $w_7, w_8$  and  $w_9$ . We calculate now the output-relevant services at distance two, which is  $X(X(w_O)) = X(\{w_6, w_7, w_8, w_9\})$ .  $X(\{w_6, w_7, w_8, w_9\})$  can be simply computed as the union of  $X(w_6) \cup X(w_7) \cup X(w_8) \cup X(w_9)$  which is  $\{w_1, w_2, w_3\}$ . Repeating this, we finally have  $\hat{X} = \{w_6, w_7, w_8, w_9\} \cup \{w_1, w_2, w_3\} \cup \{w_I\}$ , where  $w_I$  is the dummy service omitted in Fig. 3 that provides the input concepts of the request (concepts in  $L_0$ ). Since  $w_4, w_5, w_8 \notin \hat{X}$ , these services ( $w_4=MoviesDB$  Service,  $w_5=GeoLoc$  WS,  $w_8=Zip$  Search) are not contributing to the goals and can be removed from the graph.

##### 4.4.2 Interface Dominance

Another strategy to reduce the graph size is to analyse the equivalence and dominance of some services over others in terms of the interface they offer. It is very frequent to find services from different providers that offer similar services with overlapping interfaces. In scenarios like this, it is easy to end up with large composition graphs that make very hard to find optimal compositions in reasonable time. One way to attack this problem is to analyse the *interface dominance* between services in order to find those that are equivalent or better than others in terms of the interface they provide.

*Definition 10:* Given a concept in a composition graph  $G$  ( $c \in G$ ), we denote  $\Phi(c)$  as a function that returns the set of *output-relevant* services for concept  $c$ :

$$\Phi(c) = \{w = \{In_w, Out_w\} \in G \mid Out_w \otimes \{c\} = \{c\}\} \quad (3)$$



For instance,  $\Phi(\text{Payment})$  in Fig. 3 is  $\{w_8, w_9\}$  since  $Out_{w_8} \otimes \{\text{Payment}\} = \{\text{Payment}\}$  and  $Out_{w_9} \otimes \{\text{Payment}\} = \{\text{Payment}\}$ , that is, concept *Payment* is matched by an output from  $w_8$  (*PaymentID*) and for an output from  $w_9$  (*PayNum*).

**Definition 11:** A service  $w_i = \{In_{w_i}, Out_{w_i}\} \in G$  is *input-equivalent* ( $In_{w_i} \equiv In_{w_j}$ ) with respect to a service  $w_j = \{In_{w_j}, Out_{w_j}\} \in G$  in the composition graph  $G$  if:

$$\bigcup_{c_i \in In_{w_i}} \{\Phi(c_i)\} = \bigcup_{c_j \in In_{w_j}} \{\Phi(c_j)\} \quad (4)$$

That is, the set of sets defined by the union of  $\Phi(c)$  for each input concept  $c$  of each service must be equal. This definition formalises the idea of input equivalence of two services of the composition graph regarding the relation between their inputs and the services that match those inputs. That means that two services  $w_i$  and  $w_j$  of the graph are input equivalent if the services that provide the inputs of both services are the same.

**Definition 12:** A service  $w_i = \{In_{w_i}, Out_{w_i}\} \in G$  is *input-dominant* ( $In_{w_i} \succ In_{w_j}$ ) with respect to a service  $w_j = \{In_{w_j}, Out_{w_j}\} \in G$  in the composition graph  $G$  if:

$$\bigcup_{c_i \in In_{w_i}} \{\Phi(c_i)\} \subset \bigcup_{c_j \in In_{w_j}} \{\Phi(c_j)\} \quad (5)$$

Thus, informally, a service is input-dominant if it only needs a subset of the information required by the dominated service to be invoked. For example, in Fig. 3,  $w_7$  is input-dominant respect to  $w_6$ , since  $\{\{w_1, w_2\}\} \subset \{\{w_1, w_2\}, \{w_I\}, \{w_3\}\}$ .

**Definition 13:** Given a concept in a composition graph  $G$  ( $c \in G$ ), we denote  $\Psi(c)$  as the function that returns a set of input concepts in  $G$  that are matched by  $c$ , that is, there exists an arc from  $c$  to  $c'$  in  $G$ .

$$\Psi(c) = \{c' \mid (c, c') \in G\} \quad (6)$$

**Definition 14:** A service  $w_i = \{In_{w_i}, Out_{w_i}\} \in G$  is *output-equivalent* ( $Out_{w_i} \equiv Out_{w_j}$ ) respect to a service  $w_j = \{In_{w_j}, Out_{w_j}\} \in G$  in the composition graph  $G$  if:

$$\bigcup_{c_i \in Out_{w_i}} \Psi(c_i) = \bigcup_{c_j \in Out_{w_j}} \Psi(c_j) \quad (7)$$

That is, two services are output-equivalent if their outputs are matched to the same input concepts in the graph, which means that their outputs can be consumed in the same way by the same services in  $G$ .

**Definition 15:** A service  $w_i = \{In_{w_i}, Out_{w_i}\} \in G$  is *output-dominant* ( $Out_{w_i} \succ Out_{w_j}$ ) respect to a service  $w_j = \{In_{w_j}, Out_{w_j}\} \in G$  if:

$$\bigcup_{c_i \in Out_{w_i}} \Psi(c_i) \supset \bigcup_{c_j \in Out_{w_j}} \Psi(c_j) \quad (8)$$

Therefore, one service is *output-dominant* with respect to another service of the graph  $G$  if their outputs match the same inputs of the same services in the composition graph but the dominant service also provides additional outputs to the same or different services.

**Definition 16:** a service  $w_i = \{In_{w_i}, Out_{w_i}\}$  is *interface-equivalent* to a service  $w_j = \{In_{w_j}, Out_{w_j}\}$  ( $w_i \equiv w_j$ ) if  $In_{w_i} \equiv In_{w_j}$  and  $Out_{w_i} \equiv Out_{w_j}$ , that is, both are *input-equivalent* and *output-equivalent*.

**Definition 17:** A service  $w_i$  *interface-dominates* a service  $w_j$  ( $w_i \succeq w_j$ ) if the first dominates the second in at least one aspect (input-dominant or output-dominant) and is at least equivalent in the other aspect. Formally,  $w_i \succeq w_j$  if  $(In_{w_i} \succ In_{w_j} \wedge Out_{w_i} \succ Out_{w_j}) \vee (In_{w_i} \equiv In_{w_j} \wedge Out_{w_i} \succ Out_{w_j}) \vee (In_{w_i} \succ In_{w_j} \wedge Out_{w_i} \equiv Out_{w_j})$ .

This dominance definition can be generalised to include more features, such as preconditions, effects, or non-functional properties like QoS:

**Definition 18:** A service with multiple properties  $w_i = \{P_{w_i}^1, P_{w_i}^2, \dots, P_{w_i}^n\}$  where  $P_{w_i}^1$  are the inputs,  $P_{w_i}^2$  the outputs and the rest of parameters are different properties, dominates another service  $w_j$  ( $w_i \succeq w_j$ ) with parameters  $P_{w_j} = \{P_{w_j}^1, P_{w_j}^2, \dots, P_{w_j}^n\}$ , if  $\forall k \in \{1, \dots, n\} P_{w_i}^k \succeq P_{w_j}^k \wedge \exists k \in \{1, \dots, n\} P_{w_i}^k \succ P_{w_j}^k$ .

The interface dominance optimisation allows to reduce the size of the composition graph by substituting the original services of the graph by abstract interfaces that capture the functionality of the dominant or equivalent services. By minimising the graph size we improve the performance of the search algorithms since they only explore a reduced search space. Once the search is performed and the optimal composition workflow is generated, a post-processing step can be used to replace the abstract service interfaces with specific implementations using the original dominant / equivalent services or by combinations of dominated services that satisfy the same functionality of the dominant service.

## 4.5 Optimal Composition Search

The previous optimisations are intended to reduce the composition graph but keeping the same functionality. The next step is to perform a search over the graph to find the best composition among all the possible compositions that satisfy the input/output request. The search can be designed to optimise different criteria, such as the number of services, the execution path length or QoS properties. Typically, the search over the graph can be done forwards or backwards. In the first case, the composition starts from the inputs of the request (first layer), selecting invocable services until the goal outputs are obtained, whereas the second case starts with the goal outputs (last layer), selecting relevant services for the outputs until a composition that can be invoked with the initial inputs is found.

Formally, the composition search can be modelled as a state-transition system, where the problem is divided into a set of states and transitions between states [35]. A state transition system is defined as a 3-tuple  $\Sigma = (S, A, \gamma)$ , where:

- $S = \{s_1, s_2, \dots\}$  is a finite set of states.
- $A = \{a_1, a_2, \dots\}$  is a finite set of actions.
- $\gamma : S \times A \rightarrow S$  is a state-transition function.

Using the concept of the state-transition system, the state space search problem can be defined as  $P = \{\Sigma, s_0, G\}$ , where  $s_0 \in S$  is the initial state and  $G \subseteq S$  is a set of goal states.

The state-transition system  $\Sigma$  allows the search to navigate through the set of states applying different actions, where each action may be associated to a cost that we want to minimise. The state representation may vary depending on the strategy used. Typically, in the case of the backward search, the state will contain the information of the unsatisfied concepts at each state, starting with the goal outputs. The goal then is to find a succession of actions  $\langle a_1, a_2, \dots, a_n \rangle$  with the minimum cost that leads from the initial state, where unsatisfied concepts = goal outputs, to the goal state, where unsatisfied concepts =  $\emptyset$ , that is, there are no unsatisfied concepts and the composition is invocable. The available transitions between states are given by the applicable actions to each state, i.e., the output relevant services that can be selected to resolve all the unsatisfied concepts.

Given a composition graph  $G = (V, E)$  as defined previously, where  $V = W \cup C$  is the set of vertices which are the services and the concepts (inputs/outputs) of the graph, the state-transition system  $\Sigma$  for the (backward) composition problem is defined as follows:

- $S \subseteq 2^{|C|}$  where  $C$  is the set of all concepts in the composition graph, i.e., a state is a set of concepts of the graph,  $s = \{c_1, \dots, c_n\}$ .
- $A \subseteq 2^{|W|}$  where  $W$  is the set of services in the composition graph, i.e., an action is a set of services from the graph,  $a = \{w_1, \dots, w_n\}$ .
- $\gamma(a, s) = (s - \bigcup(\Psi(c_i) \mid c_i \in Out(a)) \cup In(a))$ , i.e., the application of an action  $a = \{w_1, \dots, w_n\}$  to a state  $s = \{c_1, \dots, c_n\}$  generates a new state where all concepts that are matched by the outputs of the services of the actions are removed, and the inputs of the services of the actions are added as the new unsatisfied concepts. Functions  $In(a)$  and  $Out(a)$  return the union of the input concepts and the union of the output concepts of the services in  $a$  respectively.

The initial state  $s_0$  of the backward composition problem  $P = (\Sigma, s_0, G)$  is defined as  $s_0 = In_{w_0}$ , i.e., the input concepts of the output dummy service. For example, in Fig. 3, the initial state is  $s_0 = \{i_{18}, i_{19}\}$ . On the other hand, there is just one goal state  $G = \{s_g = \emptyset\}$ , i.e., the goal state is reached when there are no unsatisfied concepts in the composition.

The efficiency of the search can also be improved using *search optimisations* depending on the search strategy followed. These optimisations can be applied to the available actions for each state by pruning actions that lead to dead-ends, actions that are equivalent, or actions that are dominated (cannot lead to a better solution).

## 5 REFERENCE IMPLEMENTATION

We developed a reference implementation of the integrated graph-based composition framework that is based

on two main components: iServe [2], a service warehouse with advanced discovery support which provides the service registry and takes care of the matchmaking and service discovery activities, and ComposIT [22], which is in charge of the graph-based composition part.

Fig. 5 depicts the architecture of the system. In a nutshell the composition process is carried out as follows. When a composition request is sent to the system through the Web UI, ComposIT starts computing the composition graph with all the relevant services for the request. To this end, all the relevant services are discovered layer by layer using the fine-grained I/O logic-based discovery support provided by the Semantic Discovery Engine of iServe. This engine relies on the *Service Manager* and the *KB Manager* to retrieve the relevant services using semantic reasoning capabilities. During the composition graph generation, ComposIT also makes intensive use of the *KB Manager* in order to carry out concept level matching and consequently figure out how the inputs and outputs of the services obtained can be connected. Once the composition graph is generated, ComposIT applies the *backward pruning* and the *interface dominance* optimisations to reduce the graph size. These optimisations are applicable using only the information contained in the graph, and thus there is no need to interact with the discovery component. Finally, an optimal search is performed over the graph using a backward A\* algorithm that extracts the optimal composition from the graph.

In the next sections we shall cover in more detail the inner workings of iServe and ComposIT respectively.

### 5.1 iServe

iServe [2], see right hand-side of Fig. 5, is a service warehouse whose functionality includes the core service registry anchored on Linked Data principles, semantic reasoning support, advanced discovery functionality, and further analysis components able to assist in automatically locating and generating semantic service descriptions out of Web resources. For the purposes of this work we have essentially exploited the registry and discovery functionality.

The service discovery functionality builds on top of the Storage Access Layer, which is in charge of managing the registry's data that includes Service descriptions, related documents and the corresponding Ontologies. This layer essentially provides a RDF/S and OWL storage and reasoning support, document storage, as well as basic crawling facilities to automatically obtain referenced Ontologies. RDF/S and OWL storage and reasoning support is delegated to dedicated engines which are accessed by means of the SPARQL 1.1 standard. Therefore, the reasoning capabilities depend largely on the actual configuration of the store. Concretely, the discovery infrastructure contacts the Service Manager to list services given basic criteria such as the input and output types provided, and the KB Manager to

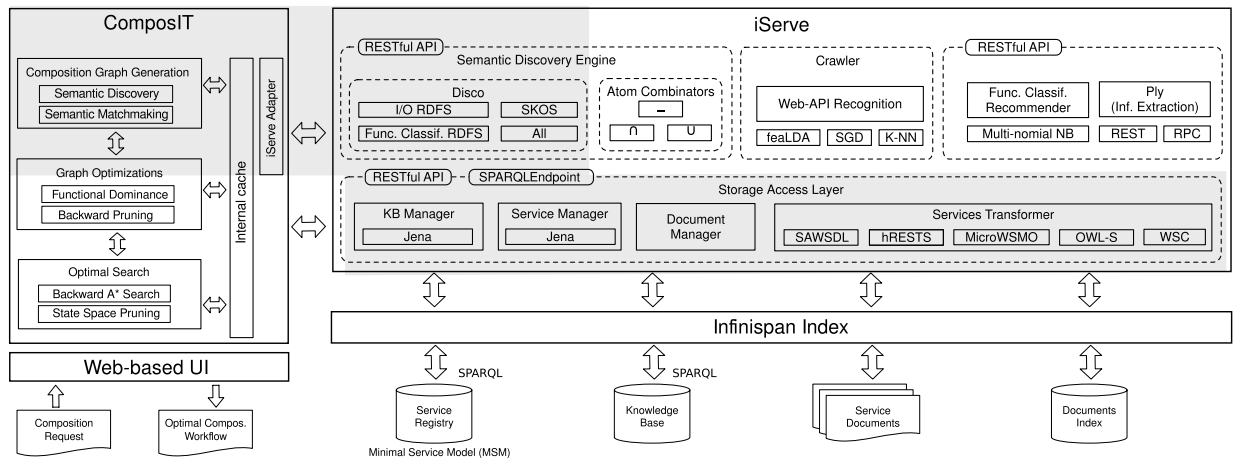


Fig. 5. CompositIT / iServe architecture

obtain concepts, properties, and their sub or super concepts. Depending on their implementation Service and KB Managers combine internal indexes with *SPARQL* queries issued to the triple store by means of Jena.

Services are imported to iServe using a range of transformation engines able to import service descriptions in a variety of formalisms including SAWSDL, WSMO-Lite, OWL-S, and MicroWSMO. These plugins generate descriptions expressed in terms of a simple RDF/S model, Minimal Service Model (MSM) [2], which essentially captures the intersection of existing service description formalisms. By means of these transformations iServe provides an homogeneous description for services that were originally annotated using heterogeneous means.

Given that, as we saw in Section 4, the response time of the overall composition is highly dependent on the performance of the service discovery and concept matchmaking tasks, we extended iServe with various implementations of the Service and Knowledge Base Managers. We tested different configurations to study their individual performance and the overall impact on composition response times. In particular, we used the following configurations:

- 1) *SPARQL D/M*: pure *SPARQL* Discovery / Matchmaking where all interactions with the Service and Knowledge Base managers are directly implemented as *SPARQL* queries. This is the typical approach of discovery engines and was the original implementation of iServe.
- 2) *Index. D/SPARQL+Cache M*: I/O service discovery is based on an index. We additionally used herein an intermediate cache at the level of the concept matcher in order to avoid issuing recurrent *SPARQL* queries.
- 3) *Full Indexed D/M*: both service discovery and concept matchmaking relied on local indexes pre-populated at load time (and updated with writes). In this configuration, service discovery and concept matchmaking do not need to issue any *SPARQL* query to the backed.

## 5.2 CompositIT

CompositIT [22], depicted in the left hand-side of Fig. 5, is the semantic Web service composition engine we rely on. It implements all the different graph-based composition phases of the framework described in Sec. 4. The semantic service discovery and matchmaking mechanisms, which originally were directly implemented internally, are delegated to iServe by means of integration adapters implemented for the purposes of this work. CompositIT nonetheless uses an internal cache and an index to efficiently recover the information of the generated composition graph. It is worth to note that the architecture supports the deployment of multiple, distributed iServe instances to provide different endpoints that can be used by CompositIT in the composition phase by aggregating the results of the registries at the CompositIT API level. Indeed, since the services to contemplate at composition time are identified by the remote registry and we just use them directly, composing this set of services out of just one API call or several calls in parallel (one per registry) is a trivial change. The overall response time analysis would still remain unchanged, and would have an upper-bound determined by the slowest registry. This also applies to other third-party discovery engines as long as they support fine-grained I/O discovery queries as described in Sec. 4.2. The integration of these third-party registries could be achieved by developing interface adapters (with capabilities to retrieve input and output relevant services) which could be plugged in to the system, keeping the generation of the composition graph isolated from the concrete registries used.

The generated composition graph can contain different compositions with the same or different length (number of layers) and with different number of services depending on the services that have been selected to generate the needed data. Among the different combinations that can be obtained, the goal of CompositIT is to find the shortest service composition with the minimum number of services. For this purpose, CompositIT searches for the optimal composition by carrying out a heuristic search

based on the A\* algorithm [36]. This search was implemented using Hipster4j [37] to identify a minimal subset of the services from the graph that satisfy the request (in terms of inputs and outputs). Note that multiple compositions can be extracted from the composition graph since there may be different services that generate outputs of the same concept.

## 6 EVALUATION

In this section we present a quantitative evaluation of our approach. The purposes of the evaluation are: 1) measure the scalability of the approach with many services; 2) study the impact of the discovery on the overall composition performance and 3) compare the performance with different optimisations.

In order to perform a standard and comparable evaluation, we selected the Web Service Challenge 2008 (WSC'08) service datasets. These datasets allow us to measure the scalability with an increasingly large set of services (from 158 to 8,119 services). Services were imported to iServe using an specific transformer plugin which translates each service description in the WSC'08 XML format into MSM, and the XML concept taxonomy into an equivalent OWL representation. iServe is responsible of identifying, loading and reasoning with the ontologies used in the service descriptions. Data types of the input and outputs of service descriptions are linked to their corresponding semantic concepts through the *modelReference* property of the MSM, which points to the concepts defined in the transformed OWL model.

Experiments were run under Ubuntu 10.04 64-bit on a PC with an Intel Core 2 Duo E6550 at 2.33GHz and 4 GB of RAM. OWLIM-Lite 5.3 with OWL Horst reasoning was chosen in iServe as the RDF triple store for the semantic registries and deployed within Tomcat 7.

TABLE 1  
Characteristics of the WSC'08 datasets.

Dataset	#Serv.	#Con.	#Serv.Sol.	Length
WSC'08 01	158	1,540	10	3
WSC'08 02	558	1,565	5	3
WSC'08 03	604	3,089	40	23
WSC'08 04	1,041	3,135	10	5
WSC'08 05	1,090	3,067	20	8
WSC'08 06	2,198	12,468	40	9
WSC'08 07	4,113	3,075	20	12
WSC'08 08	8,119	12,337	30	20

Table 1 shows the characteristics of each WSC'08 dataset. The number of services and concepts in the ontology of each dataset are shown in columns *#Serv.* and *#Con.* respectively. The quality of the solutions is based on the number of services and the length (i.e., number of layers) of the composition. The optimal quality of solution for each dataset (according to the WSC'08 competition) are shown in columns *#Serv.Sol.* and *Length*.

Experimentation was done using the configurations explained in Sec. 5 with one instance of iServe in order to

measure the effect of the Discovery/Matchmaking over the whole composition process. Results with each configuration are shown in Table 2. The second column shows the size (number of services) of the resulting composition graph for each dataset. The next columns show the time taken to generate the composition graph (*G. time*) in seconds and the number of *SPARQL* queries generated during that process. The last three columns show the size of the graph after the graph-based optimisations, the time of the composition search (graph optimisations + optimal A\* backward search) and the number of services and length of the optimal composition found. Note that the backward optimal search does not depend on the configuration selected since it only uses the information in the composition graph.

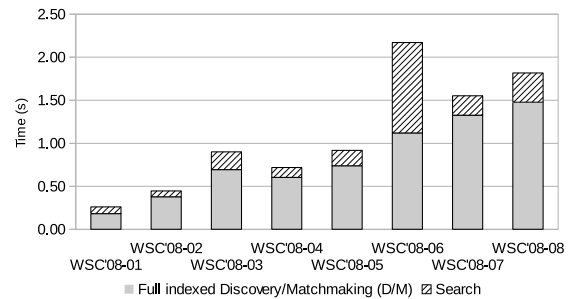


Fig. 6. Graph generation time vs Search time for the Full Indexed Discovery/Matchmaking configuration.

The analysis of these results reveals that the discovery and matchmaking phases take most of the time of the composition, even using the optimal configuration (*Full Indexed D/M*) to avoid the latency of the *SPARQL* queries. This is graphically represented in Fig. 6. This figure shows the overall composition time for each dataset including the relative time of the *Full Indexed D/M* (blue bar) and the *Composition Search* (red bar). The *Full Indexed D/M* takes 77% of the total composition time on average. This percentage is even higher (about 99%) if the discovery and matchmaking are not optimised using indexes and cache. In other words, as anticipated by the complexity analysis presented earlier, discovery and matchmaking are responsible for the majority of the computation that needs to be performed to compose services. Optimising both phases is thus fundamental.

The comparison of the scalability of the three configurations with respect to the number of services is shown in Fig. 7. As can be seen, directly querying the backend (see *SPARQL D/M*), which is the approach followed by most discovery engines, rapidly becomes prohibitively slow taking 1,656 seconds (i.e., 27.6 min) in the largest dataset. Indeed, the generation of the composition graph requires computing every semantic match between all inputs and outputs as well as discovering relevant services at each layer. Doing so leads to issuing thousands of *SPARQL* queries. This can be dramatically improved using a discovery index and a local cache for the matchmaking system as can be seen in the second

TABLE 2  
Evaluation results with different Discovery/Matchmaking (D/M) configurations with the WSC'08 datasets

Dataset	G. size	Discovery/Matchmaking (D/M)						Composition		Sol. (serv./length)
		1) SPARQL D/M		2) Index. D/SPARQL+Cache M		3) Full Indexed D/M		G. size (opt)	Comp. time (s)	
		G. time (s)	#SPARQL	G. time (s)	#SPARQL	G. time (s)	#SPARQL			
WSC'08-01	35	28.52	3256	5.67	624	0,18	0	13 (-37%)	0.08	10/5
WSC'08-02	35	63.30	7349	11.76	1830	0,38	0	13 (-37%)	0.07	5/3
WSC'08-03	105	262.80	36619	20.05	3184	0.69	0	40 (-38%)	0.21	40/23
WSC'08-04	44	136.20	13828	21.12	3481	0.60	0	25 (-57%)	0.12	10/5
WSC'08-05	97	333.60	41148	26.05	4417	0.74	0	52 (-54%)	0.18	20/8
WSC'08-06	189	1051.20	93682	48.21	8511	1.12	0	75 (-40%)	1.05	42/7
WSC'08-07	124	1183.20	120881	35.76	6376	1.33	0	70 (-56%)	0.23	20/12
WSC'08-08	121	1656.00	89518	78.00	15844	1.48	0	58 (-48%)	0.34	30/20

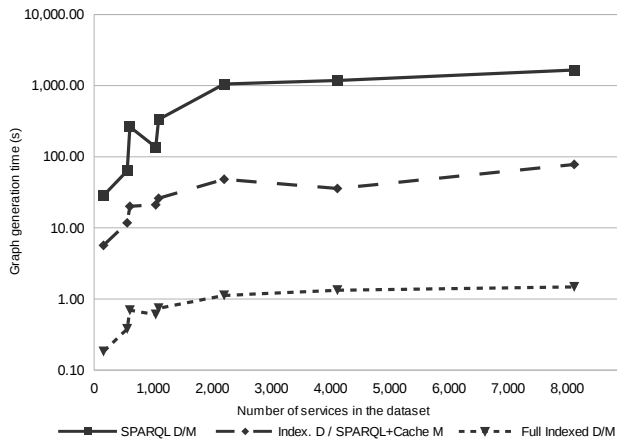


Fig. 7. Composition time for different configurations.

configuration. In this case, almost every composition is calculated in less than a minute. The generated *SPARQL* queries in this case are reduced by up to 91% (for the WSC'08-3 dataset) leading to a significant performance improvement. Although such an improvement can be enough to solve the smaller datasets in a few seconds, the latency of the *SPARQL* queries still remains a bottleneck for bigger datasets like the WSC'08-08 dataset that still require evaluating 15,844 *SPARQL* queries for generating the composition graph in 78 seconds. Our tests show, however, that the full indexed configuration allows solving the largest problems very fast by avoiding the evaluation of *SPARQL* queries at composition time. This configuration entails the derived need for service registries to additionally calculate and maintain the indexes. Doing so, nonetheless, enables performing very efficient composition over remote 3rd party controlled service registries akin to what can be obtained by the fastest composition engines in the unrealistic scenarios where all services are available and pre-loaded in memory. Additionally, indeed, using those indexes allows service registries to provide highly efficient discovery for a controlled set of queries, while retaining the ability to offer fully flexible yet less efficient discovery support.

We have also evaluated our framework with the WSC'09-10 datasets. Results show a similar scalability

behaviour with the number of services for each configuration. Moreover, our approach is able to solve all the datasets with optimal results, which are shown at <https://wiki.citius.usc.es/composit:wsc09>.

## 7 CONCLUSIONS

In this paper we have presented a theoretical analysis of service composition in terms of its dependency with service discovery. Driven by this analysis we have defined a formal integrated graph-based composition framework anchored on the integration of service discovery and matchmaking within the composition process. We have devised a reference implementation of this framework on the basis of two pre-existing separate components, namely *iServe* and *ComposIT*. This reference implementation has been used to empirically study the impact of discovery and matchmaking on service composition, and we have provided three different configurations with varying performance. Our empirical analysis shows that, indeed, typical approaches followed by discovery engines cannot serve as a suitable basis to support efficient service composition as they lead to prohibitive execution times. We have also shown, though, that with the adequate interface granularity and indexing, discovery engines can support highly efficient composition akin to that obtained by the fastest composition engines without having to assume to local availability and in-memory preloading of service registries.

This work proves the scalability and flexibility of our proposal and provides insights on how integrated composition systems can be designed in order to achieve good performance in real scenarios, where service registries and composition frameworks are likely to be distributed and controlled by diverse organisations.

## ACKNOWLEDGMENT

This work was partly supported by the Spanish Ministry of Economy and Competitiveness (MEC) under grant TIN2011-22935, and by the COMPOSE European Project (FP7-ICT-317862). Pablo Rodríguez-Mier is supported by an FPU Grant from the MEC (ref. AP2010-1078) and was also partially funded by Pedro Barrié de la Maza Foundation (2013).

## REFERENCES

- [1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services*, ser. Data-Centric Systems and Applications. Springer, 2004.
- [2] 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.
- [3] B. Srivastava and J. Koehler, "Web Service Composition - Current Solutions and Open Problems," in *ICAPS 2003 workshop on Planning for Web Services*, 2003, pp. 28–35.
- [4] J. Rao and X. Su, "A Survey of Automated Web Service Composition Methods," in *Semantic Web Services and Web Process Composition*, vol. 3387, 2004, pp. 43–54.
- [5] S. Dustdar and W. Schreiner, "A survey on web services composition," *Int. J. of Web and Grid Services*, vol. 1, no. 1, pp. 1–30, 2005.
- [6] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.
- [7] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, pp. 38–45, 2007.
- [8] M. Carman, L. Serafini, and P. Traverso, "Web Service Composition as Planning," in *ICAPS 2003 Workshop on planning for web services*, 2003, pp. 1636–1642.
- [9] 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.
- [10] 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.
- [11] S. A. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the Eight Int. Conf. on Principles and Knowledge Representation and Reasoning (KR-02)*, 2002, pp. 482–496.
- [12] E. Sirin, B. Parsia, and J. Hendler, "Template-based composition of semantic web services," in *In AAAI Fall Symposium on agents and the semantic web*, 2005, pp. 85–92.
- [13] S. Kona, A. Bansal, M. B. Blake, and G. Gupta, "Generalized Semantics-Based Service Composition," *IEEE Int. Conf. on Web Services*, pp. 219–227, 2008.
- [14] Y. Yan, B. Xu, and Z. Gu, "Automatic Service Composition Using AND/OR Graph," in *10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services*, 2008, pp. 335–338.
- [15] M. Aiello, N. Benthem, and E. Khoury, "Visualizing Compositions of Services from Large Repositories," in *10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services*. IEEE, 2008, pp. 359–362.
- [16] W. Nam, H. Kil, and D. Lee, "Type-Aware Web Service Composition Using Boolean Satisfiability Solver," *2008 10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services*, vol. 1, pp. 331–334, 2008.
- [17] K. Raman, Y. Z. Y. Zhang, M. Panahi, and K.-J. L. K.-J. Lin, "Customizable Business Process Composition with Query Optimization," *2008 10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services*, 2008.
- [18] M. Shiao, J. Fladmark, and B. Thiell, "An Incremental Graph-based Approach to Automatic Service Composition," in *2008 IEEE Int. Conf. on Services Computing*. IEEE, 2008, pp. 397–404.
- [19] P. Hennig and W.-T. Balke, "Highly Scalable Web Service Composition Using Binary Tree-Based Parallelization," *2010 IEEE Int. Conf. on Web Services*, pp. 123–130, 2010.
- [20] S. Oh, D. Lee, and S. Kumara, "Web service planner (WSPR): an effective and scalable web service composition algorithm," *Int. Journal of Web Services Research*, vol. 4, no. 1, pp. 1–22, 2007.
- [21] M. Klusch, "Overview of the S3 Contest: Performance Evaluation of Semantic Service Matchmakers," in *Semantic Web Services*. Springer, 2012, pp. 17–34.
- [22] P. Rodriguez-Mier, M. Mucientes, J. Vidal, and M. Lama, "An Optimal and Complete Algorithm for Automatic Web Service Composition," *Int. J. of Web Services Research (IJWSR)*, vol. 9, no. 2, pp. 1–20, 2012.
- [23] C. Pedrinaci and J. Domingue, "Toward the Next Wave of Services: Linked Services for the Web of Data," *Journal of Universal Computer Science*, vol. 16, no. 13, pp. 1694–1719, 2010.
- [24] C. Pedrinaci, J. Domingue, and A. P. Sheth, "Semantic web services," in *Handbook of semantic web technologies*. Springer, 2011, pp. 977–1035.
- [25] M. Klusch and P. Kapahnke, "OWLS-MX3: an adaptive hybrid semantic service matchmaker for OWL-S," in *Proceedings of 3rd Int. Workshop on Semantic Matchmaking and Resource Retrieval*, 2009.
- [26] M. Klusch and P. Kapahnke, "iSem: Approximated reasoning for adaptive hybrid selection of semantic services," in *The semantic web: Research and applications*. Springer, 2010, pp. 30–44.
- [27] D. Chu, J. Han, J. Li, and Y. Zhao, "XSSD: A Fast Hybrid Semantic Web Services Discovery Method," in *Int. Conf. on Computer Technology and Development, 3rd (ICCTD 2011)*, 2011.
- [28] J. Peer, "Web Service Composition as AI Planning – a Survey," University of St. Gallen, Switzerland, Tech. Rep. March, 2005.
- [29] O. Hatziz, D. Vrakas, M. Nikolaidou, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "An Integrated Approach to Automated Semantic Web Service Composition through Planning," *Transactions on Services Computing*, pp. 1–14, 2011.
- [30] A. Bansal, M. B. Blake, S. Kona, S. Bleul, T. Weise, and M. C. Jaeger, "WSC-08: Continuing the Web Services Challenge," in *10th IEEE Conf. on E-Commerce Technology and the Fifth IEEE Conf. on Enterprise Computing, E-Commerce and E-Services*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 351–354.
- [31] F. Lécué, E. Silva, L. F. Pires, and F. Lecue, "A framework for dynamic web services composition," *2nd ECOWS Workshop on Emerging Web Services Technology*, pp. 59–75, 2007.
- [32] E. G. da Silva, L. F. Pires, and M. van Sinderen, "Towards runtime discovery, selection and composition of semantic services," *Computer Communications*, vol. 34, no. 2, pp. 159–168, Feb. 2011.
- [33] 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.
- [34] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic Web Service Composition with a Heuristic-Based Search Algorithm," in *IEEE Int. Conf. on Web Services (ICWS)*, 2011, pp. 81–88.
- [35] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice*. Elsevier, 2004.
- [36] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions of Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.
- [37] P. Rodriguez-Mier, A. Gonzalez-Sieira, M. Mucientes, M. Lama, and A. Bugarin, "Hipster: An Open Source Java Library for Heuristic Search," in *9th Iberian Conf. on Information Systems and Technologies*, 2014.

**Pablo Rodríguez-Mier** is a PhD student at CiTIUS, Universidade de Santiago de Compostela, Spain. His research interests include heuristic search and automatic Web service composition.

**Dr. Carlos Pedrinaci** Dr. Carlos Pedrinaci is a Research Fellow of the Knowledge Media Institute at The Open University, UK. He holds a PhD in Artificial Intelligence from the University of the Basque Country (Spain). His research interests include Web Science, Semantic Web, and Service Science. Carlos has been actively involved in the standardization of Semantic Web Services technologies as part of the W3C SAWSDL Working Group and, recently, the Linked USDL initiative.

**Dr. Manuel Lama** is Associate Professor at CiTIUS, Universidade de Santiago de Compostela. His research interests focuses on discovery and composition of Web services, semantic annotation, and process mining.

**Dr. Manuel Mucientes** is Associate Professor in Computer Science and Artificial Intelligence within the CiTIUS of the Universidade de Santiago de Compostela. His current research interests are evolutionary computation, robotics, and Web services.