# Answering Complex Requests with Automatic Composition of Semantic Web Services

Brahim Batouche
*Public Research Center Henri Tudor, Luxembourg*
*brahim.batouche@tudor.lu*

Yannick Naudet
*Public Research Center Henri Tudor, Luxembourg*
*yannick.naudet@tudor.lu*

Frédéric Guinand
*University of Le Havre, France*
*frederic.guinand@univ-lehavre.fr*

***Abstract*—Today, there is a growing need for user to be able to express, and get answers to more complex requests, those including multiple functionalities, conditions, constraints and objectives. Complex requests including multiple functionalities only can not usually be answered with one single Web service.**

**As multiple services are needed, the problem is then to find good combination using the available services. This paper contributes to answering this issue. It focuses on the problem of semantic Web services composition to answer such requests.**

**We propose an automatic composition algorithm designing the answering composition. The algorithm takes into account all suitable composition structures. The set of answering composition is modeled as a graph, which supports the selection of the best composition according to the request constraints and objectives.**

***Keywords*-complex web request; composition of web services; semantic web services.**

## I. INTRODUCTION

Electronic commerce (e-commerce) presents an important average gain for enterprises of different domains: tourism, transport, etc. For this reason the e-commerce has a growing interest in Web services to publish its products. Additionally, the Web services relative simplicity gives information providers and users easy access to new content. They are now widely available on the Web. For instance in 2009, the number of Web pages for e-tourism (electronic tourism) has been estimated to 65.2 billions, and represents in Europe 25.7% of the market [1]. As a side effect of this success, customer requirements become more and more complex, such that finding a single service fitting the specific needs of a user is unlikely.

Let us consider someone wanting to organize a stay in Rome. Using natural language, he/she could express his/her wish as: " *I want to visit Rome for a week-end, I would like to go there by airplane and to stay in an hotel*". Such request has two functionalities: flight and hotel booking. A functionality is associated to a service and each one accepts inputs and produces outputs. The departure and destination cities of the flight are the input parameters of the flight service. The localization and the date are the input parameters of the hotel service. These parameters have to be coherent, e.g. the arrival date in Rome has to be the same as the starting date of the Hotel reservation.

The request is processed in two steps: a first step for determining the services able to answer the different functionalities, i.e., the services allowing to book a flight and to book an hotel, and a second step for actually executing the functionalities, i.e., actual booking of flight and hotel. The first step is achieved by calling informative services (IS), while the second one is performed by active services (AS). IS output is used as input by AS.

This paper is an extension of [2]. We propose an algorithm for automatically finding all candidate compositions answering a complex request, without a priori knowledge of the composition structure. When the request does not formally specify any chaining between the request elements, the algorithm finds suitable composition structures based on the available services. The structure of the composite service depends obviously on the request, but also of the available services. The underlying problem is not trivial because there are many possible combinations of services, as well as many composition structures.

Services composition is useful in many different domains, e.g., tourism, transport, multimedia, etc. Some of them involve a dynamic environment where at any time events can affect previously computed compositions. The proposed algorithm can compensate services failures by finding a new executable composition when this happens.

In Section I-A and I-B we define respectively the research context and the useful definitions. In Section II, we present the state of the art and our contributions. In Section III, we formalize the problem elements: service, request and answering composition. In Section IV, we present how to describe semantically Web services and complex request. In Section V, we present the model of answering compositions. Section VI present our algorithm for automatic construction of a composition. Section VII is concerned with an experimentation results on which we assess our algorithm. Section VIII present specific cases of request resolutions. Finally, we conclude in Section IX.

### A. Background

Complex requests such as the example presented here before, cannot be answered with one single service. Different processing steps are required. [3] decomposes the request resolution into: service presentation, service translation, composition generation, composition evaluation, and composition execution. In [4], the request processing is decomposed into: Web services discovery, planning, execution, and optimization. In [5], the request treatment follows three steps: discovery, plan generation and plan optimization. It appears that the choice of the resolution steps depends on the considered point of view: [4], [5] take the customer point of view, while [3] considers the provider point of view.

In this paper, we take the customer point of view and we consider both the composition design time and runtime to define our resolution steps. Design time can be decomposed in three steps: services discovery, composition design (or planning [4]) and selection of compositions. Runtime can be decomposed into execution and adaptation steps [6]. We propose a general resolution process for answering complex request made of five steps: (1) description of request functionalities, constraints and objectives; (2) services discovery: to find suitable services; (3) composition design: to determine how the services can be composed together to answer the request, and to design the corresponding composite service; (4) selection of answer(s): to select the composited services fulfilling the best the request objective(s) and respecting the request constraint(s), and finally (5) composition execution.

Figure 1 illustrates the different steps from the user request to the actual execution of the different services. The outputs of one step are the inputs of the following one. This paper focuses on steps (1) and (3), the step (2) being not considered as it is widely studied (see e.g., [7]). We also discuss the dynamic composition case, occurring when either services are faulty on when they can no more be invoked at execution time.

The request solving is decomposed into service discovery and service selection. When the request is complex we design the answer using composition of existing Web service. The automatic composition provides the answering composition set. Then the selection of the best composition can require an optimization method.

Automatic composition is based on automatic service discovery and combines the different discovered services to answer a complex request. Functional parameters in the request are used to find services, while non-functional parameters (request constraints and objectives) are exploited to select the best matching services.

### B. Overview and Definitions

Web services composition consists in two steps: design time (design of composition answering a request) and runtime (composition execution). At design time, two steps can

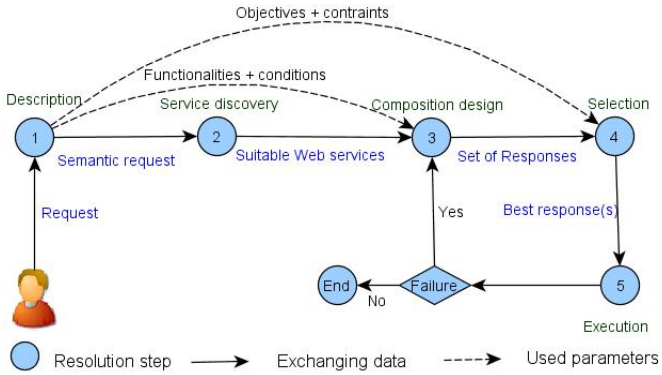


Figure 1. Complex request processing principle.

also be considered: (1) the search of possible compositions regarding the available services; and (2) the selection of best compositions based on request objective and constraint. In the following we provide related definitions.

*1) Composition Design:* The composition design can be performed in three ways: automatically, semi-automatically and manually. Moreover, the composition can be abstract or executable.

Manual composition is a combination of services directly specified directly by a designer. Semi-automatic composition determines the composing services and their control flow progressively by interrogating the user. Automatic composition determines the composing services and their control flow progressively without user interaction.

An executable composition allows services invocation and can be used as a composite service. An abstract composition comprises only the functionalities and their control flow, without giving any execution possibility.

Following our requirements, we consider executable services only. Indeed, the automatic composition requires being able to invoke some services at design time: in particular informative services, which will provide input data for other services. Additionally, the composition itself needs to be executable.

The composition of Web service is designed to achieve a specific goal. This goal is achieved by composing many services, and then building a new service, called *"composite service"*. The composite service can be modeled as a composition path, tree, organization of agents, chromosome, etc. The building of a composite service involves different composition structures such as: sequence, choice, switch, while, split (starting parallelism) and split-joint (ending parallelism), see Figure 2. Many other terms are used for naming composition structures like: control flow, execution plan, and planning. In this paper these terms will be used as synonyms.

*2) Composition Runtime:* The composition runtime can be done in static or dynamic environment, the composition in dynamic environment is named " dynamic composition"

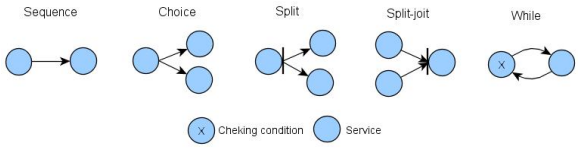


Figure 2. Composition structures illustration

while in static environment it is named "static composition". The dynamic composition takes into account possible execution failures (service breakdown) and provides some executable alternative composition(s). This case is not considered in static composition [6].

## II. State of the Art

Since our work concerns three items: complex request description, automatic composition design and answering composition model, this section presents the existing works in these areas and highlight our contributions.

### A. Complex Request Description

Several works try to describe semantically the complex request. In [8], a complex request is presented as a set of inputs, outputs and conditions, which are formalized in description logic $\mathcal{ALN}$. The works [9], [10] present a complex request as an abstract composite Web service and use service profile of OWL-S [11]. The service model can be used to describe the internal structure. However, decomposing a request in a connected set of services, estimating data and control flow, is not obvious for users.

The request formalization used in [8], [10], [9] is somehow limited. [8] considers the request I/O and condition, but the use of $\mathcal{ALN}$ as a formalization language makes it difficult to use with existing services standards implementations (using, e.g., OWL-S). The works [10], [9] use OWL-S description, but they do not consider all the request elements of formula 1, such as constraint and objective.

We modelled our request ontology to take into account all the elements specified in the request formalization (formula 1), and with the objective of keeping full advantage of existing works on semantic Web services, such as the matchmaking service OWL-MX [12]. Additionally, the user request needs to be formalized in such a way it can directly be used to find matching services.

### B. Composition Models

Before analyzing the existing works, we present our requirements. According to our objectives, the composition model should fulfill the five following points: (1) represent a search space for optimization, specifying functional and non-functional parameters of each composition. (2) support all possibles composition structures: sequence, choice, switch, while, split and split-joint, (3) allow the composition invocation, (4) support dynamic environment and dynamic composition: allowing to search for alternative compositions when failures occur during execution, (5) allow the translation of compositions into existing Web services description languages.

A model for the composition of Web services represents a set of services and their links, achieving a specific set of functionalities. According to researchers point of view and requirements, the composition is modeled by different mathematical representation: Petri nets [13], [14], [15], [16], [17], Directed Acyclic Graph [18], [19], [5], Workflows [20], [3], Situation Calculus [21], UML diagrams [22], Finite State System [23] or Multi Agents System (MAS) [24].

*1) Petri nets:* Are the widely used to model services composition, because they allow to model the steps and events in distributed system and consider the compositions structures. The composition is modeled with a set of places, transitions and tokens: places correspond to services, transitions correspond to input/output exchanges between services, tokens correspond to atomic operations [14]. Functional and non-functional parameters of services are specified in the places of Petri nets, and not specified in the transitions. We can not model services having values of parameters according to other service, because non-functional parameters are specified in the places. For example, when the hotel reservation cost changes according to the travel agency service, the hotel service can have different costs (non-functional parameter value), which cannot be specified in the same place [13], [14], [15], [16], [17]. Non-functional parameters could be specified as a transition weight vector, but Petri nets are not specifically designed to achieve this goal.

*2) Directed Acyclic Graphs:* DAG can be used to represent the execution order of services. The graph nodes represent the services and the arcs represent the data flow between services. The composition is modeled as a path. The functional parameters are specified in the node and the non-functional parameters are specified on the arcs. This structure allows to overcome the limit of Petri nets models. However, unlike Petri nets, DAG have some restrictions that prevent representing all the structures illustrated in Figure 2. For instance, DAG acyclic constraints is incompatible with the loop/while structure. To overcome this limit DAGs can be adapted for different composition structures [18], [19], [5].

*3) The following synthesis can be made about the other approaches: Workflow* presents a composition as a sequence of tasks and the message flow between services. The workflow semantic [20] allows to facilitate the interoperability of heterogeneous Web services. Workflows can model composition structures, but do not allow representing multiple compositions. *UML activity diagrams* is a descriptive model, inspired from Petri nets model [22]. The UML diagram is used for describing one compositions, and not a set. *Situation calculus* is efficient to construct the composition

in dynamic environment, because it represents the actions, and the situation as a sequence of actions. However, it is not obvious to exploit situation calculus to evaluate a composition. Additionally, it cannot be used to model multiple compositions, since it builds one composition step by step [21]. *Finite state system (FSS)* represents a set of states and transitions. States represent the services. Transitions consider all possible actions to execute a composition. However, FSS represents a composition as a sequence of services, and it is not well suited to consider other structures such as split or split-joint. Last, with *Multi Agents System (MAS)* [24] a services composition can be represented by a set of agents. Roles of agents corresponds to functionalities of services; input, output, precondition and effect. The MAS can be used to model the organization of compositions, but it provides a model that cannot be used as an optimization search space, because the non-functional parameters are not considered.

Each of the aforementioned models, has advantages and drawbacks for a specific requirement, but fails to meet all requirements listed at beginning of this section. Among the possible models we have listed, only the DAG is suitable, because it can be easily adapted to our requirements of supporting any composition structure and composition execution. Adaptation of the other models remains a priori too complex, without any foreseen benefit.

### C. Automatic Composition Design

As defined in I-B1, automatic composition allows designing a composition without interrogating the user, defining the different Web services components and their composition structures. Some works [25], [26], [27], [15], [28], [29], [16], [24], [17] try to achieve this goal.

In [25], rules are used to generate a sequence of Web services, from the relations between them. Sequences are the only composition structures considered in that work.

In [26] a flooding algorithm is used. It first looks for services matching a request input. Then the algorithm progresses step by step by finding next services having input matching the output of previously selected services. The progression finishes when a selected service output matches with the request output.

An architecture for automatic Web service composition is proposed in [27], which according to the authors, allows fast composition of OWL-S services. The proposed approach uses implicitly a flooding algorithm. However, while authors provide interesting ideas for the design of the composite service and automating service invocation, they only consider sequences of services.

[15] proposes an algorithm based on matrix-equation approach and the provided compositions are modeled with Petri Nets. The compositions are built to answer a request. During the building, the method allows to check the reachability of the composition, by checking accessibility of states from initial state. This approach is also useful to verify the validity of the built composition by other methods, but it does not allow the evaluation of the composition.

Oh, Lee, and Kumara [29] present an AI-planning algorithm of Web services composition, called WSPR. The algorithm relies on the request input and output, and then achieves the composition in two steps: (1) It computes the cost to achieve the composition, beginning with the request input. (2) This cost is used as guidance, to minimize the length of the sequence of services. This approach allows designing and selecting the composition at the same time, which it is not possible for any request, as discussed in section VIII-A. Additionally it considers only the sequence structure.

The algorithm in [28] builds a composition graph from a given request. It identifies first the input and output of the request and searches for a matching service. If none can be found, a service having only a matching output is selected and recursively, subsequent services having output matching with the input of the latter service and input matching with the request input are sought. The algorithm ends when a sequence of services starting with the request input and ending with its output is found, or when the set of available services has been visited. While the algorithm allows the composition execution, it is limited to sequences structures, such as [26]. The algorithm in [28] uses an inverse direction of built [26], and it has been designed to minimize the number of compositions, because generally an input can correspond to services having different outputs. The method of [26] is suitable to have a large set of compositions.

[16] presents an algorithm for "configuration" of compositions and selection of the best composition according to the services cost. The compositions are modeled with Petri Nets and are selected by considering the non-functional requirements. This approach allows to configure and select the composition at the same time, as in [29].

In [24], a MAS is used to answer a user request using automatic composition. The composition is based on a reasoning loop to determine the composition plans answering the request. An agent is limited to an OWL-S service and its functional parameters describing the agent role. The agents collaborate to provide the composition needed. The proposed reasoning algorithm allows to detect the composition plan according to the service semantic, but it does not consider all composition structures.

[17] builds the composition relying on semantic of service input/output. An hypergraph representation, as in [30], is used to determine the suitable services, and then to model the composition with Petri Nets. Using the semantic of services I/O ensures the coherency of composed services. The use of an hypergraph allows the determination of suitable compositions according to functional parameters, without considering non-functional parameters. However, it considers only the sequence structure. According to our requirements, it is

suitable to model all compositions without selection. The latter is reported in the compositions optimization step.

To automate the composition design, considering our requirements, the dedicated algorithm has to allow determining the composition structures and executing the composition. For this, we have to rely on the semantic of services input and output (as in [17]), and also on the semantic of composition structures.

### D. Contributions

Our contributions to the Web services composition research field are respectively:

1) To formalize the different concepts of the problem: Web service, complex request and composition of Web service.
2) To propose an ontology (OWL-CR) describing semantically the complex request.
3) To model the composition by considering all requirement cited at beginning of section II-B.
4) To propose an algorithm solving a complex request with automatic composition, which considers any composition structures and detects them automatically.
5) To determine when the composition requires an optimization. When this is the case, we explain how to represent the composition set as a search space for optimization. In the other case, we explain how to select the best composition.

The existing studies address the composition problem only partially, and none of them considers all the points cited above. Additionally, no formalization of the problem can be found in the state of the art. In the following section, we clearly state the problem to solve.

## III. Formalization

The problem space of automatic service composition concerns three main elements: a complex Web request to solve, a set of available Web services and service compositions that are form the services to answer the request. In this section, We formalize these elements.

### A. Complex Request

*Definition 1:* A complex request illustrates a service a user asks for, following a specific goal, for which he/she specifies both functional and non-functional parameters. It can be represented as a set of functionalities on which conditions can be expressed, and a set of constraints and objectives expressing non-functional parameters.

The complex request can then be written as a three-tuple: $\mathcal{R} =< \mathcal{F}_\mathcal{R}, \mathcal{D}, \mathcal{NF}_\mathcal{R} >$, where $\mathcal{F}_\mathcal{R} = (F_{R_i})^T$ is the vector of functionalities, $^T$ being the transposition operator, where each functionality $F_{R_i}$ is mandatory or optional and has as input and output respectively $I_{F_{R_i}}$ and $O_{F_{R_i}}$. $\mathcal{D} = \{d_1, ..., d_r\}$ is a set of conditions, where a condition refers to a request functionality input ($I_{F_{R_i}}$) or output ($O_{F_{R_i}}$); and $\mathcal{NF}_\mathcal{R}$ is the set of non-functional parameters.

Lets first write $\mathcal{I}_\mathcal{R} = \{I_{F_{R_i}}\}$ and $\mathcal{O}_\mathcal{R} = \{O_{F_{R_i}}\}$, denoting respectively the input and output set of the request, $\mathcal{NF}_\mathcal{R}$ can be defined as $\mathcal{NF}_\mathcal{R} =< C, B, \lambda >$, where $\mathcal{C} = \{c_1, \cdots, c_m\}$ is the set of constraints, $\mathcal{B} = \{obj_1, ..., obj_k\}$ is the set of objectives, and $\lambda$ is the set of importance levels associated to objectives. Finally, this leads to the following formula:

$$\mathcal{R} =< \mathcal{I}_\mathcal{R}, \mathcal{O}_\mathcal{R}, \mathcal{D}, \mathcal{C}, \mathcal{B}, \lambda > \quad (1)$$

Conditions $\mathcal{D}$ differ form constraints $\mathcal{C}$ in that they have to be verified by services answering a part of the request or for instantiating the input parameters of service (e.g., the depart and destination cities for a transport service), while constraints allow filtering the set of answering services or data obtained by the Informative Service ($IS$), or the composition obtained by the automatic composition process [2].

Objectives of $\mathcal{B}$ will have to be minimized (e.g., cost, time) or maximized (e.g., performance, availability). In a specific context, an objective will be minimized (e.g., the service execution $time$ is minimized) and maximized in another context (e.g., the leisure activity $time$ is maximized).

Constraints and objectives can refer to data, e.g., the flight price must not exceed 200 Euro ($C_d$), or, minimize the flight price ($B_d$). They can refer to services, e.g., the service price must not exceed 5 Euro ($C_s$) (minimization); or to the composition, e.g., the travel price must not exceed 3000 Euro ($C_c$) (minimization). We write accordingly $\mathcal{C} =< C_d, C_s, C_c >$ and $\mathcal{B} =< B_d, B_s, B_c >$ . It should be noticed that a composition constraint $C_c$ can refer to one or multiple optional functionality(s) (e.g., if the price exceeds 2500 Euro, then cancel the sports activity service to reduce the price).

The importance levels $\lambda$ are specified by the user. They can concern one single functionality (e.g., the transport price is more important than its quality.) or the whole request (e.g., the price is more important than the quality, globally). In the first case, we write $\lambda_{i,l}$, where $i$ and $l$ are respectively the objective and functionality indexes. In the latter case, it is simply written $\lambda_i$. The set of objectives importance levels can be written: $\lambda = \{\lambda_{i,l}, \lambda_i\}$. Additionally $\lambda$ can be a single value (e.g., $\lambda_1 = 0.4, \lambda_2 = 0.6$) or an interval (e.g., $\lambda_1 \in [0, 0.4[, \lambda_2 \in [0.6, 1[$ ). We have $\sum_{i=1}^{i=k} \lambda_i = 1$, and $\forall l, \sum_{i=1}^{i=k} \lambda_{i,l} = 1$. We note that a request functionality can be answered with a service or a composition of services. The latter can contain different types of services, informative, active or the both.

Finally, the request conditions can induce different kinds of dependency between functionalities. We distinguish three functionalities dependency types: no dependency, one-to-

one dependency and global dependency. The two latter are defined in Definition 2. Different strategies will be applied depending on their kind, as will be explained in section VIII-B.

*Definition 2:* One-to-one dependency in the request means that all dependencies existing between request functionalities concern at most two of them, and each functionality has at most one dependency.

*Definition 3:* Global dependency in the request means that there exists at least one functionality having more than one dependency.

### B. Web Services

The literature proposes several definitions of Web services. Basically: *a Web service is a collection of protocols and standards used for exchanging data between applications* [31]. More specifically: *Web services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces* [32]. Very specifically: *a Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols* [33].

Mathematically, we formalize a Web service as a five-tuple:

$$\mathcal{S} :=< \mathcal{ID}, \mathcal{FP}, \mathcal{NF}, \mathcal{CS}, \mathcal{NS} >, \quad (2)$$

where $\mathcal{ID}$ denotes the service identifier and access (e.g., service URI), $\mathcal{FP}$ denotes its functional parameters, $Input, Output, Precondition, Effect$ (IOPE); $\mathcal{NF}$ denotes its non-functional parameters (e.g., service name, service price, etc.), $\mathcal{CS}$ denotes the composition structure (i.e., the service control flow); and $\mathcal{NS}$ is the set of ID inner services (or operations). The existing Web service languages try to annotate this five-tuple by describing them syntactically or semantically.

A Web service is modeled according to a specific goal. For this reason, there exist three Web service models [34]: black box, white box and semi-transparent box. The black box describes only the service functionalities, $\mathcal{FP}$, the semi-transparent box describes $\mathcal{FP}$ and the service composition structures, $\mathcal{CS}$, and the white box details all inner services, $\mathcal{NS}$. According to [34], the white box is hardly useable, because it implies formalizing the service program in too much details.

Solving a complex request implies modeling both the existing services and the answering compositions. Regarding existing services, the black box model is privileged because knowing the functional parameter of services is enough to discover matching services. For the answering compositions, which can be used as an optimization search space, the mixed use of white box and semi-transparent box (white-semitransparent box) is suitable. This allows describing all inner services $\mathcal{NS}$ and composition structures $\mathcal{CS}$.

There are two types of service: informative service ($IS$) and active service ($AS$). The first provides some data (e.g., list of things) and does not modify its database after invocation. The second performs an action and modifies its database after its invocation (e.g., flight booking). There are significant differences between IS and AS regarding their usage in our resolution approach. Indeed, the IS will be executed at composition time to aggregate the provided information in the composition, whereas the AS will be executed at composition runtime, i.e., after the optimization has been performed, and after the user has selected its preferred composition among the best compositions proposed by the optimization method. The service can answer a request functionality partially or completely.

### C. Composition of Web Services

Web services compositions is the combination of multiple service operations. These operations can follow a specific invocation order (i.e., a control flow or composition structure). A service composition can be formalized as a triple:

$$\mathcal{SC} :=< \mathcal{OP}, \mathcal{CF}, \mathcal{E} >, \quad (3)$$

where $\mathcal{OP}$ denotes the set of service operations $\{op_1, ..., op_j\}$, $\mathcal{CF}$ denotes the set of control flow constructs (or composition structures) and $\mathcal{E} \subseteq (\mathcal{OP} \cup \mathcal{CF}) \times (\mathcal{OP} \cup \mathcal{CF})$ are edges connecting operations and control flow constructs [35]. For example, if we consider a DAG of composition $G =< N, A >$, where $N$ is the set of nodes and $A$ the set of arcs, we have: $\mathcal{SC} :=< \mathcal{OP} = N, \mathcal{CF} = \{Arc\ sequence\}, \mathcal{E} = A >$.

In order to define the set $\mathcal{CF}$, we first review the most common Web service composition languages: OWL-S and BPEL4WS [36]. The existing languages for Web services composition allow to model different composition structures in different ways. Taking the most commonly known, we observed the following. The structures modeled by OWL-S are: "sequence", "any-order", "if-then-else", "choice", "while", "until", "split" and "split-joint". Differently, BPEL4WS [36] uses: "sequence", "switch", "while", "pick" and "flow". A mapping between the two representations involves two operators: equivalence (e.g., "choice" is equivalent to "pick"); decomposition (e.g., the "flow" structure in BPEL4WS can be decomposed into two structures of OWL-S: "split" and "split-joint"; "switch" is set of imbrication "if-then-else"). So, "while", "until", and "any-order" can be described by other structures.

In order to insure a compatibility with the different representations and keeping a generic approach, we focus

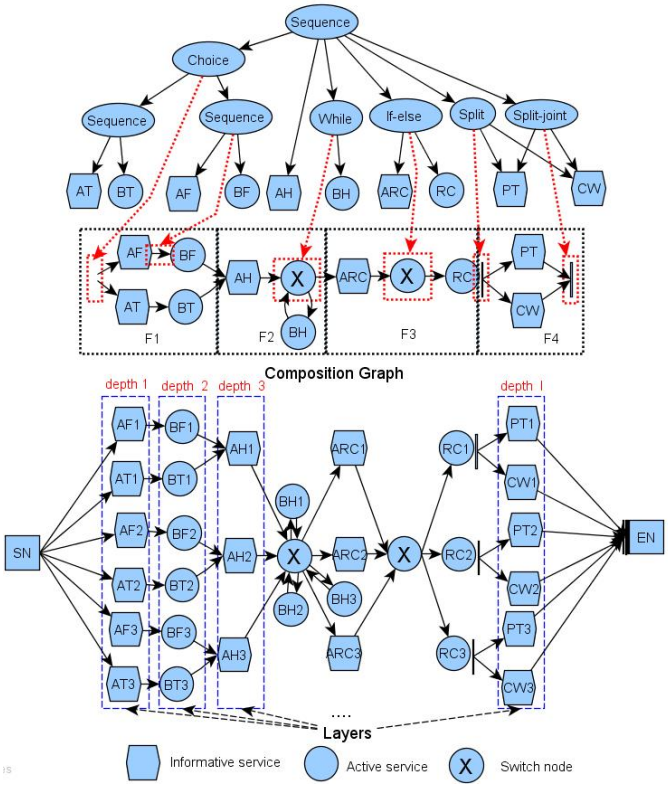


Figure 3. Service composition illustration: tree, flow and graph, where, AT means Available Train, AF means Available Flight, BT means Book Train, BF means Book Flight, AH means Available Hotel, BH means Book Hotel, ARC means Available Rentals Car, RC means Rent Car, PT means Plan Touristic map, CW means City Weather, SN start node and EN end node.

on elementary structures (sequence ($sq$), choice ($ch$), switch ($sw$), split ($sp$), split-joint ($sj$)), from which many others can be modeled. Consequently, $\mathcal{CF}$ is defined as:

$$\mathcal{CF} = \{sq, ch, sw, sp, sj\}, \quad (4)$$

*1) Composition Structures Illustration:* A composition may comprise several different structures, which can themselves contain combinations of structures. A tree representation helps understanding and visualizing the composition: the leaves are service operations, the nodes and the root are the compositions structures. A flow representation corresponds to the reading of the composition tree by following a depth-first search.

Let us consider a composition answering the following request: *"I want to travel from City A to City B, reserve several hotel rooms in destination city where each booking is billed separately, rent a car for six people, have the weather forecast and plan for the destination city"*. The Figure 3 shows the composition tree, the corresponding composition path and the composition graph.

*2) Characteristics of Composition Structures :* In the following, we describe the different composition structures, and we provide a syntax for them.

Sequence "$\rightarrow$": This structure defines a total order between services. There are two ways to detect the order: (1) checking the match between services IOPE (Input, output, precondition and effect); (2) checking the dependency between the services answering the question: *which service cancels the other when it is canceled?*

Choice "+" (or-split): This structure represents a choice between several services that have a same functionality. $Choice(A, B_1, B_2, \cdots, B_k) \equiv (A \longrightarrow B_1) \vee (A \longrightarrow B_2) \vee \cdots \vee (A \longrightarrow B_k)$, knowing that service $A$ precedes all services $\{B_i\}$ which have the same functionality.

If-then-else "$\otimes_c$": This structure is the classical condition branching. It allows a conditional service execution or choice of services input parameters values. This structure checks a request condition, and controls a service if at least one of its functional parameters corresponds to a request condition predicate. The switch structure is based on If-then-else structure, because it represents an imbrication of this latter. In the following, switch and if-then-else terms are used as synonyms.

Split "$\vdash$": This structure indicates a simultaneous start of multiple services (or services chains) that can be parallelized. The parallelized services have the same predecessor and provide different types of outputs. Each parallelized service can start a new sub-path in the composition. All services chains starting at a split will be executed in parallel and ended with a split-joint. $\text{Split}(A, B_1, B_2, \cdots, B_k) \equiv (A \longrightarrow B_1) \wedge (A \longrightarrow B_2) \wedge \cdots \wedge (A \longrightarrow B_k)$.

Split-joint "$\dashv$": This structure ends a parallel structure, where different composition paths belong to a same "split" and the last services $\{B_i\}$, in parallel chains, have the same successor $A$. Split-Joint $(B_1, B_2, \cdots, B_k, A) \equiv (B_1 \longrightarrow A) \wedge (B_2 \longrightarrow A) \wedge \cdots \wedge (B_k \longrightarrow A)$, where services $\{B_i\}$ end the parallel composition paths. It is possible that services from a same split do not end with the same Split-joint.

Any-Order "$\odot$": This structure represents a random invocation of services. It is not elementary because it means the choice between all possible alternatives, i.e., it can be expressed using choice and sequence structures: $A \odot B \equiv (A \longrightarrow B) + (B \longrightarrow A)$. At execution time, such structure can be replaced by a sequence or a parallel structure [37]. I can be noticed that if the number of services involved in the any-order structure is large, the replacement by a parallel structure may be very costly, since it is a combinatorial enumeration.

While "$\oplus_c$" and until "$\ominus_c$ ": These structures are used for iterative service invocation, and they are not elementary because they can be constructed with if-then-else and sequence structures: $\oplus_c(A) = \otimes_c \longrightarrow A \wedge A \longrightarrow \otimes_c$ and $\ominus_c(A) = A \longrightarrow \otimes_c \wedge \otimes_c \longrightarrow A$.

The Table I summarizes the detection rules of composition

| Composition structure | Detection rule |
|---|---|
| Sequence $(A, B)$ | $M(O_A, I_B) \leq \epsilon \vee (M(O_A, \mathcal{O}_\mathcal{R}[i]) \leq \epsilon \wedge M(\mathcal{I}_\mathcal{R}[i+1], I_B) \leq \epsilon)$ |
| Choice $(B_i, B_j)$ | $M(O_{B_i}, O_{B_j}) \leq \epsilon \wedge A \rightarrow \{B_i, B_j\}$ |
| If-then-else $(A)$ | $dom(a_i) = O_A$ |
| Split $(A, B_i, B_j)$ | $M(O_{B_i}, O_{B_j}) > \epsilon \wedge A \rightarrow \{B_i, B_j\}$ |
| Split-joint $(B_i, B_j, A)$ | $M(O_{B_i}, O_{B_j}) \leq \epsilon \wedge \{B_i, B_j\} \rightarrow A \wedge n.id(B_i) \equiv n.id(B_j)$ |

Table I
DETECTION RULE OF ELEMENTARY COMPOSITION STRUCTURES,

where, $I, O$ are respectively input,output; A,B are Web service; $\mathcal{I}_\mathcal{R}, \mathcal{O}_\mathcal{R}$ are respectively the request input/output; $a_i$ is a predicate of request condition ; $n.id(B_i)$ denotes the split identification, where service $B_i$ belong, $M$ the matching level and $\epsilon$ the matching threshold.

structures.

In our example in Figure 3, the AF and AT are structured with "choice" structure, because their output classes match. The AH service is controlled with "switch" structure because the request condition "*rent a car for six people*" refers to the car capacity. The latter is a property of the ARC output class. So on for the other structures.

## IV. SEMANTIC DESCRIPTION

The complex request resolution requires describing semantically the Web services and the complex request. Semantic web services provides knowledge to discover, compose and invocate services. The semantic description of a complex request provides knowledge needed to solve the request.

### A. Semantic Web Service

Semantic Web Services approaches add a semantic layer to elements of Web services, $\mathcal{ID}, \mathcal{FP}, \mathcal{NF}, \mathcal{CS}, \mathcal{NS}$ (see equation 2). This allows the automation of discovery, composition and invocation of Web services over the Web. Then, the main question is: *Which semantic layer is needed to automate service discovery, composition and invocation?*.

Existing works focus on specific elements of equation 2. For example, the WSLA (Web Service Level Agreement) project [38] adds a semantic layer on $\mathcal{NF}$, addressing service management issues and challenges in a Web services environment on SLA specification, creation and monitoring. The WSDL-S [39] language adds a semantic layer to WSDL, i.e., on $\mathcal{ID}$, $\mathcal{CS}$ and $\mathcal{FP}$. The METEOR-S (Managing End-To-End Operations for Semantic Web Services) project [40] presents a semantic annotation of WSDL, addressing the issues related to workflow process management for large-scale, complex workflow process applications in real-world multi-enterprise heterogeneous computing environments. METEOR-S is based on approaches described in [41], [18], to define an extensible ontology that describes the properties of quality of service.

According to our point of view, services discovery can be automated by adding a semantic layer to $\mathcal{FP}$; services composition can be automated by adding a semantic layer to $\mathcal{FP}$, $\mathcal{CS}$ and $\mathcal{NS}$; and service invocation can be automated by adding a semantic layer to $\mathcal{ID}$ and $\mathcal{FP}$. OWL-S is currently the most known and referred language for semantic web services. It allows more flexibility than METEOR-S by relying on a domain ontology. The process view taken by the latter not only induces some complexity in the reasoning, but also is unnecessary for the needs of our research.

The Semantic Web Services language OWL-S provides semantic layers namely for $\mathcal{ID}$ (service grounding), $\mathcal{CS}$ and $\mathcal{NS}$ (service model). $\mathcal{FP}$ is modeled by the service profile, but the semantics of IO is left to the additional use of a domain ontology, as well as the semantics of PE, which in [10] is described with SWRL rules. $\mathcal{NF}$ description is also left to the use of an additional QoS ontology.
In the following section, we present an ontology for describing complex requests and useable with OWL-S services description.

### B. The Complex Request Ontology: OWL-CR

The ontology for Complex Request we have designed is illustrated in Figure 4. The root class of OWL-CR is the class `Request`. Two main properties, namely *hasFunctionality* and *hasNFparameter*, allow respectively to formalize the different functionalities and non-functional parameters of a request.

For each functionality, a `Functionality Profile` is defined, which can be `Mandatory` or `Optional`. The optional profile is not taken into account when its consideration induces the non-respect of a constraint. Each element of the request in formula 1 has a corresponding class in the ontology. For a given request $\mathcal{R}$, the $I_{F_{R_i}}, O_{F_{R_i}}$ are input or output classes of a service domain ontology (in our case, the travel booking service ontology), the condition $\mathcal{D}$ is described using the `ruleml:Impl` class of SWRL, defining a rule. This is formalized using respectively the `hasInput, hasOutput` and `hasCondition` properties. The `parameter type` in Input and Output classes allow specifying the URI of corresponding classes in a service domain ontology modeling services characteristics in a particular domain.

The non-functional parameters, constraints $\mathcal{C}$ and the objectives $\mathcal{B}$, are respectively instances of `Constraint` and `Objective`, which can depend on the non-functional parameters of `Service`, `Data` or `Answered composition`. The importance level of an objective $\lambda$

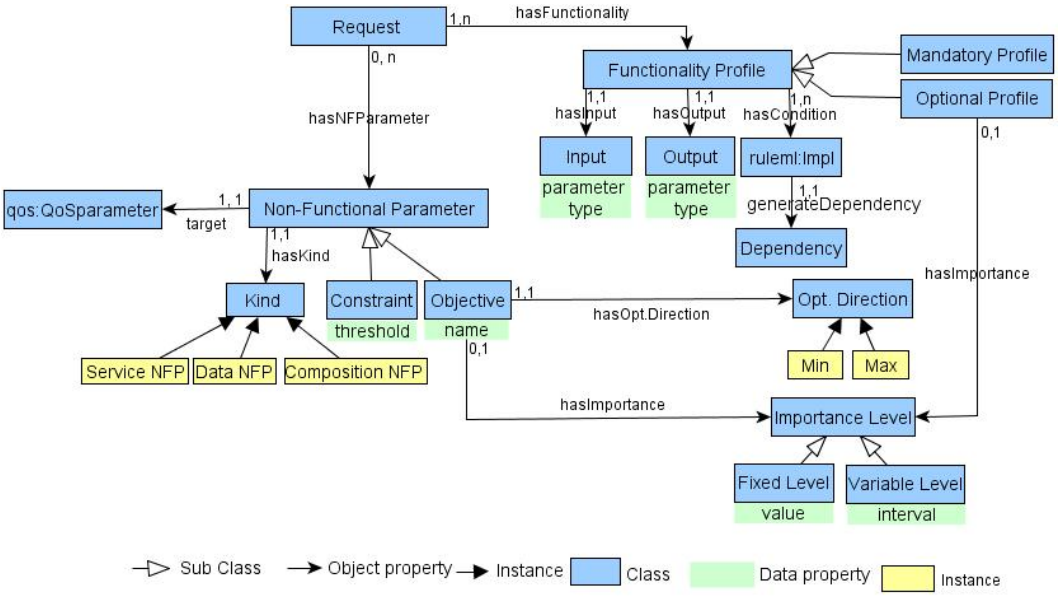


Figure 4. Web service complex request ontology (OWL-CR)

is an instance of `Importance Level`, which can be a `Fixed level` or `Variable level`. The objective has an optimization direction: `Min` or `Max`, used to specify how it has to be exploited.

The ontology OWL-CR contains different cardinalities (0,1), (0, n), (1,1) and (1,n), where $n \geq 1$. According to the OWL language capability, it has been formalized in OWL DL. OWL-CR is used as a language to describe complex requests. Practically, this means that users inputs, through a dedicated user interface, are used to construct a request description formalized in OWL-CR.

For an ideal situation, functional and non-functional parameters of the request need to be expressed using the same domain and QoS ontologies of the services to be queried. The OWL-CR ontology we have designed for expressing requests uses directly these ontologies. Moreover, it exploits SWRL to formalize the conditions.

## V. Models of Answers

The goal of this section is to present how the set of valid compositions answering a request is modeled, according to the request characteristics. The request functionality dependencies have an impact on the adapted compositions models. One can numerate three cases:

1) When no dependency is specified, the composition is modeled as a sequence following a random order and the set of compositions can be modeled as a graph.
2) In the case of one-to-one dependencies, the composition is modeled as a sequence following the specified order of dependent functionalities and the set of composition can be modeled as a graph.
3) When the dependencies are global, a composition can be modeled as a sequence and we cannot model the set of valid compositions as a graph. We detail this point in section VIII-B.

### A. Graph of Compositions

When the request contains no dependency or one-to-one dependencies, the compositions answering the request are modeled as a graph. The composition graph $G$ models the set of possible connexion between services. The complete paths starting from the first node and ending at the last node of the graph, are composition answering the request. $G$ is composed of several layers according to request functionalities. Each layer contains a set of services answering the same request functionality. The composition graph is formulated as: $G =< N, A >$, where $N$ is a set of nodes $\{n_j\}$ and $A$ is a set of arcs $\{a_i\}$. An arc is defined as $a_i =< b, e, w >$, where $b$ is the depart node, $e$ is the destination node and $w$ is a weight. Nodes $n_i$ correspond to service. $G$ has two special nodes representing the start, $sn =< I_{F_{R_1}} >$, where $I_{F_{R_1}}$ is the first element of the request input; and the end of all compositions $en =< O_{F_{R_l}} >$, where $l$ is the index of the last functionality. The composition paths of $G$ are defined by the following definitions.

*Definition 4:* Let $\Omega$ be the set of paths in a composition graph answering a request $\mathcal{R}$. We have:
$\forall x = (b \rightarrow e) \in \Omega : b = sn, e = en$

*Corollary 1:* $\forall x \in \Omega, \forall c \in C_c \cup C_d$: $x \models c_i, \exists I, M(I_{F_{R_i}}, I) \leq \epsilon, \exists O, M(O_{F_{R_j}}, O) \leq \epsilon, \forall I_{F_{R_i}} \in \mathcal{I}_{\mathcal{R}}, \forall O_{F_{R_j}} \in \mathcal{O}_{\mathcal{R}}$.

Where $\models$ is the satisfaction symbol, $I/O$ is the input/output of inner services of $x$, $C_c, C_d$ are set of con-

straints referring respectively to composition and data, $\epsilon$ is the matching threshold level and $M(y, z)$ is the matching level between $y$ and $z$.

A composition path has both functional and non-functional parameters. Functional parameters are affected to the nodes (i.e., service $I/O$), and non-functional parameters are affected to arcs. An arc weight is a vector containing request objectives values $v_{obj_i}$ for the arc destination node. Formally, let $a_{i,j} : (n_i \rightarrow n_j)$ be an arc and $w_{a_{i,j}}$ its weight, $w_{a_{i,j}} = \left( \ v_{obj_1}(n_j), \cdots, v_{obj_k}(n_j) \ \right)^T$.

The composition graph is executable when it allows invoking the compositions. The nodes then contain the communication protocol, which is specified in service grounding of OWL-S description.

The composition graph can consider all defined composition structures or only the sequence structure. Considering all composition structures is a way to expand the search space of objectives, because we consider different evaluation functions of an objective.

The evaluation function of objectives can change according to the composition structure (as detailed in table **??**). For example, the duration $(dr)$ objective has different evaluation functions: $(max(dr_i))$ for a parallel structure and $(\sum dr_i)$ for a sequence structure. For some parameters, it never changes. For example, the price $(pr)$ objective has the evaluation function $(\sum pr_i)$, for any composition structures.

When the evaluation functions of all request objectives are the same for any composition structures, we consider only the sequence structure and the composition graph is then acyclic. The latter can become cyclic if we consider additionally if-then-else structures. When we consider all possible composition structures, the obtained composition graph is called multi-structure: its paths are adapted to consider the composition structures.

In summary, considering the request functionalities have no global dependency, when the request objectives have the same evaluation function $f_{obj_i}$ for all the composition structures $(cs)$, the composition graph is acyclic. Otherwise, the graph is multi-structures.

*1) Multi-Structures Composition Graph:* In order to answer all of our requirements and in particular considering all compositions structures. We adapt the DAG to a Multi-Structures Composition Graph (MSCG). We define a node as a six-tuple:

$$n =< NT, id, URI_S, URI_{DA}, I_s, O_s >, \quad (5)$$

where $NT$ is the node type, with $NT = \{IS,\ AS,\ DA,\ SW\}$, where $IS$ is an informative service, $AS$ is an active service, $DA$ is data and $SW$ is a switch node that represents a conditional structure. $id$ is an identifier for nodes starting a parallel structure ($id = \phi$ if $n$ does not belong to a parallel structure); $URI_S$ is the service URI ($URI_S = \phi$ if $NT = DA$); and $URI_{DA}$ is the data URI ($URI_{DA} = \phi$ if $NT = IS \ \ or \ \ AS$). $I_s/O_s$ are respectively the input and output of the service represented by the node.

The formalization (5) is suitable to have short runtime complexity but unsuitable to have low memory complexity, because some elements are useful only during the composition design, such as $NT, I_s, O_s$. However, the nodes are memorized as : $n =< URI, id >$.

The switch node is used to model the request conditions $\mathcal{D}$ in each composition. It is defined as: $n_{SW} =< NT = SW, d_i, HF >$, where $d_i$ is a request condition, and $HF$ is a hash function, that is used to store reference to nodes verifying the condition. For this reason, the switch node is a kind of meta-node containing several nodes.

In the MSCG, composition structures (St) are represented as a combination of a node and arcs. We write: $St :=< N, A >$. The Table II gives the different forms for each kind of structure.

| Structures | node definition |
|---|---|
| while and until | $< SW, A_1 >$ |
| if-then-else | $< SW, \phi >$ |
| sequence, choice, split and split-joint | $< \phi, A_2 >$ |

Table II
DEFINITION GRAPHIC OF COMPOSITION STRUCTURES,

where $A_1$ is the sequence-arcs, and $A_2$ is respectively the set of sequence-arc, split-arc and split-joint-arc.

In MSCG, the arc weights must be adapted for specific cases. If the destination node type is data ($NT(n_j) = DA$), then some objectives are not defined, because non-functional parameter of service are not the same at data base property. For example, the runtime objective might be missing in data representing hotel related information. In this case, we affect a neutral value to the objective. If the type of the arc destination node is "switch" ($NT(n_j) = sw$), then $w_{a_{i,j}}$ will be affected by neutral value, because switch node has any impact in weight of composition.

As detailed in section VIII-A, the optimization can be required. If so, the MSCG is transformed to provide a suitable search space. The composition structures (if-then-else, parallel) have not any impact on optimization and are note considered. The MSCG will then be transformed by reducing these structures. The obtained composition graph is cyclic if the MSCG contains while/until structures, it is acyclic otherwise.

The split and split-joint represent a specific type of structure, and they are transformed into a node regrouping all services belonging to the parallel structure. The associated arcs are transformed into sequence arcs. The weight of the split arc is recalculated as:
Let $a_{i,j} : n_i \rightarrow n_j$ be an arc, if $a_{i,j}$ be a split type, then $w_{a_{i,j}} = \left( \ v_{obj_1}(f_{fl}), \cdots, v_{obj_k}(f_{fl}) \ \right)^T$. $a_{i,j} \leftarrow sequence$. If $a_{i,j} = splitJoin$, then $a_{i,j} \leftarrow sequence$, where $f_{fl}$ is the

evaluation function of the flow structure. Finally, the MSCG transformation and the recalculation of arc weights allow building a suitable set of answering composition (called optimization search space), considering all the composition structures.

*2) Acyclic Composition Graph:* The acyclic composition graph (ACG) contains only sequences. Its nodes can be defined as nodes of MSCG, but to simplify the representation of node, one single type of node will be used and it contains variables for data URI ($URI_{DA}$), informative service URI ($URI_{IS}$), and active service URI ($URI_{AS}$).

$$n =< URI_{DA}, URI_{IS}, URI_{AS} > \quad (6)$$

Where the service URIs, $URI_{IS}, URI_{AS}$, correspond respectively to the OWL-S description of informative and active services.

The arc weight values $w_{a_{j,j+1}}$, are calculated by considering all values of $DA, IS$ and $AS$:

$$v_{obj_i}(n_{j+1}) = v_{obj_i(DA)} + v_{obj_i(IS)} + v_{obj_i(AS)}.$$

The weight $w_{a_{j,j+1}}$ values of an arc ($a_{j,j+1} : n_j \rightarrow n_{j+1}$) are the sum of the destination node $n_{j+1}$ elements (*DA, IS, AS*). These weights will be used in the optimization step, to evaluate the composition.

## VI. Composition Design Algorithms

The goal of composition design algorithms is to build the composition graph answering the request, which will be used as an optimization search space. As explained in the previous section, when the composition design considers only sequence structures, the graph is acyclic (ACG); when all composition structures are taken into account, the graph is a MSCG. When the request functionalities dependency is global, the search space cannot be modeled as a graph. It is then modeled as a set of clusters. A cluster regroups the services matching a request functionality ($\mathcal{F}_{\mathcal{R}}[i]$), and a cluster is created for each request functionality. A request functionality can require an informative service (IS), active service (AS) or both. When an IS is used, it is executed. For each data, it generates a node is created. This node integrates the link (URI) of the informative service and the corresponding active service if it is required, and then it is affected to the corresponding cluster.

The proposed design composition algorithms, process progressively the request to build the executable composition graph. The process logic is based on the flooding algorithm and checks the composition structures. This checking is based on the characteristics of composition structures (as said in section III-C2). We present in the following two algorithms: algorithm 1 considers only sequence structures, and algorithm 2 considers all composition structure.

### A. Composition Algorithms

We name the *current layer*, $L_k$, the set of nodes in the composition graph having a same depth level, being processed by the algorithm: $L_k = \{n_i\}$. Initially $L_k$ contains the starting node $sn$ of $G$, $L_k = \{sn\}$. One step of the algorithm corresponds to the full coverage of $L_k$. The node of $L_k$ being processed during an iteration is named *current node* ($L_k[i]$), where $i$ is the node index. We denote $T$ the temporary buffer, $L$ a layer of nodes, $L_{end}$ the end layer, $L_d$ a layer of data, $d_i$ a condition and $a_i$ a statement in a condition.

*Algorithm 1: $ACG$*: Design composition algorithm (Request $\mathcal{R}$)

```
1:  L_k, L_{k+1}, L_{End} ← φ;
2:  L_k.add(sn(I_{F_{R_1}}));
3:  N.addAll(L_k); // Add all nodes of L_k;
4:  i ← 0;
5:  while i < |L_k| do
6:     L_k[i] : current node;
7:     s_j ← SelectNextServices(L_k[i]); // Algorithm 3.
8:     for s_j : NextService do
9:        AddArcSequence(L_k[i], s_j);
10:       if s_j ∉ N then
11:          if EndFunctionality[s_j] then
12:             L_End.add(s_j);
13:          else
14:             L_{k+2} ← RunInformativeService(s_j); //
                Algorithm 4.
15:             for DA_k : L_{k+2} do
16:                s_j ← Concatenation(s_j, DA_k);
17:                L_{k+1}.add(s_j);
18:             end for
19:          end if
20:       else
21:          L_{k+1}.add(s_j);
22:       end if
23:    end for
24:    N.addAll(L_{k+1}); L_k ← L_{k+1}; L_{k+1} ← φ;
25:    i++;
26: end while
27: Return ACG;
```

From a request, first fill in the current graph layer with matching services and process it (line 2 for starting and line 24 during processing). For each node in the current layer, select the next services according to their matching with functionalities $F_{R_i}$ (Algo. 1 or Algo. 2 - line 7, call Algo. 3). The set of nodes created from next services is put into the next layer. When the current node output matches with one of the $F_{R_i}$ outputs, the algorithm carries on with next node in the current layer. Otherwise, services having inputs matching the current node output are selected. Corresponding nodes are created after checking they do not already exist in the set of nodes $N$ of $G$.

*Algorithm 2:* $MSCG$: Design Composition algorithm (Request $\mathcal{R}$)

1: $L_k, L_{k+1}, T, L_{End} \leftarrow \phi$;
2: $L_k.add(sn(I_{F_{R_1}}))$;
3: $N.addAll(L_k)$;
4: $i \leftarrow 0$;
5: **while** $i < |L_k|$ **do**
6: $L_k[i]$ : current node;
7: $s_j \leftarrow SelectNextServices(L_k[i])$;
8: **for** $s_j : NextService$ **do**
9: $AddArcSequence(L_k[i], s_j)$;
10: **if** $s_j \notin N$ **then**
11: **if** $EndFunctionality[s_j]$ **then**
12: $L_{End}.add(s_j)$;
13: **else**
14: $T.add(s_j)$;
15: **if** $L_k[i] \in Split$ **then**
16: $GetInSplit(s_j, L_k[i])$;
17: **end if**
18: **end if**
19: $CheckSplitJoint(i, L_k, s_j)$;
20: $L_{k+2} \leftarrow RunInformativeService(s_j$,D);
21: **if** $|L_{k+2}| = 0$ **then**
22: $SW \leftarrow CheckCondition(s_j, \mathcal{D})$;
23: **if** $SW \neq null$ **then**
24: $T.add(SW)$;
25: **end if**
26: **else**
27: $L_{k+1}.addAll(L_{k+2})$;
28: **end if**
29: **else**
30: $CheckSplitJoint(0, L_k, s_j)$;
31: **end if**
32: $ChechSplit(L_k[i], T)$;
33: $L_{k+1}.addAll(T)$;
34: $T \leftarrow \phi$;
35: **end for**
36: $N.addAll(L_{k+1})$; $L_k \leftarrow L_{k+1}$; $L_{k+1} \leftarrow \phi$; $L_{k+1} \leftarrow \phi$;
37: i++;
38: **end while**
39: $Return \;\; MSCG$;

An arc-sequence is created between the current node and each of the next nodes. When a selected service is an IS, it is invoked to obtain the data it provides before creating the arc (Algo. 1-line 14, Algo 2-line 20, call to Algo 5). Once the data are obtained (filling the data layer $L_d$), the algorithm creates an arc-sequence between the node of the service and each one of the data nodes. The selected service node is then replaced by the set of data nodes.

### B. Sub-Branches Used in Composition Algorithms

*Algorithm 3:* SelectNextServices (Service $s_i$)

1: **if** $M(Output_{s_i}, Output_{F_{R_i}}) > \epsilon$ **then**
2: **if** $M(Input_{s_i}, Input_{s_j}) \leq \epsilon$ **then**
3: $s_j$ selected;
4: **if** $M(Output_{F_{R_{i+1}}}, Output_{s_j}) \leq \epsilon \wedge |I_{\mathcal{R}}| = i + 1$ **then**
5: $EndFunctionality[s_j] \leftarrow true$;
6: **end if**
7: **end if**
8: **else**
9: **if** $M(Input_{F_{R_{i+1}}}, Input_{s_j}) \leq \epsilon$ **then**
10: $s_j$ selected;
11: **if** $M(Output_{F_{R_{i+1}}}, Output_{s_j}) \leq \epsilon \wedge |I_{\mathcal{R}}| = i + 1$ **then**
12: $EndFunctionality[s_j] \leftarrow true$;
13: **end if**
14: **end if**
15: **end if**
16: $Return \;\; selected \;\; s_j$;

When a next node has been newly created, i.e., the service corresponding to the node, is not selected already (Algo. 2, Algo. 5, Line 10), the algorithm checks the existence of a condition. This is the case if the output of the service represented by the node corresponds to one of the request conditions, or if the class of data represented by the node contains predicates used in a condition (Algo. 6). In this case, we create a SW-node and link it to the node by an arc-sequence. The node following the SW-node is then selected according to the first node output.

*Algorithm 4:* Run informative service (Service $s$)

1: $L_d \leftarrow \phi$;
2: **if** s: informative Service **then**
3: $L_{data} \leftarrow Runs(s)$;
4: **end if**
5: **for** $i = 1, |L_{data}|$ **do**
6: $AddArcSequence(s, L_{data}[i])$;
7: **end for**
8: $Return \;\; L_{data}$;

*Algorithm 5:* Run informative service (Service $s$, Condition $\mathcal{D}$)

1: $L_d \leftarrow \phi$;
2: **if** $s$: informative Service **then**
3: $L_{data} \leftarrow Runs(s)$;
4: **end if**
5: $SW \leftarrow CheckConditionData(L_{data}[0], \mathcal{D})$;
6: $L_{data}.add(SW)$;
7: **for** $i = 1, |L_{data}|$ **do**
8: $AddArcSequence(s, L_{data}[i])$;
9: **if** $SW \neq null$ **then**
10: $AddArcSequence(L_{data}[i], SW)$;

```
11:    end if
12: end for
13: Return  L_data;
```

*Algorithm 6:* CheckCondition (Node $n$, Condition $D$ )

```
1: for i = 1, |L| do
2:    if (n is service) then
3:       if (a_1 ∨ a_2 ∨ ... ∨) ∈ class(Output_s) ∨ QoS_s then
4:          SW ← CreatSwitchNode(d_i);
5:          AddArcSequence(L[i], SW);
6:       end if
7:    else
8:       if (a_1 ∨ a_2 ∨ ... ∨) ∈ DataClass(n) then
9:          SW ← CreatSwitchNode(d_i);
10:      end if
11:   end if
12:   T ← T ⋃ SW;
13:   if (d_1 ∨ d_2 ∨ ... ∨ d_n) ∈ class(Input_s) then
14:      AddArcSequence(SW, s);
15:   end if
16: end for
17: Return  SW;
```

When all $F_{r_i}$ have been covered, the next node is affected to the end layer (Algo. 2, Line 12). Otherwise, it is put into the next layer for (Algo. 1, Line 21), and for Algo. 2, it is put into temporary buffer, and later into the next layer (Line 33), after checking the split structure. The checking of split-structures is performed when the temporary buffer is full, containing all nodes matching the current node. The checking of split-joint-structure is performed when the next node is selected. Hence, the algorithm checks split-joint structure before the split structure. The process checks the existence of a split-joint structure starting from next node (Algo. 7). If it is selected from $N$ of $G$, it is possible to find a node which can precede the next node. In this case, a complete check is performed (Algo. 2, Line 30), otherwise only a partial check is necessary (Algo. 2, Line 19). The complete check considers all nodes of the current layer. The partial check considers a current node and current layer nodes which have not been yet processed.

*Algorithm 7:* CheckSplitJoint (SercieIndex $i$, NextLayer $L_{k+1}$, Service $s$)

```
1: IsSplitJoint ← false;
2: if M(Input_s, Output_{L[i]}) ≤ ε ∧ L[i] ∈ Split then
3:    for m = i + 1, |L| do
4:       if M(Input_s, Output_{L[m]}) ≤ ε ∧
         SameSplit(L[m], L[i]) then
5:          IsSplitJoint ← true;
6:          AddArcSplitJoint(L[i], L[m], s);
7:          GetOutSplit(L[m]);
8:       end if
9:       if (M(Input_s, Output_{L[m]}) > ε ∧ L[m] ∈ Split
         then
10:         if SameSplit(L[m], L[i]) then
11:            GetInSplit(s, L[i]);
12:         end if
13:      end if
14:   end for
15:   if IsSplitJoint then
16:      GetOutSplit(L[i]);
17:   end if
18: end if
```

The algorithm 7 creates a split-joint-arc when it detects nodes, $L[i], L[m]$, having the same succeeding node ($s$: ends the parallel structure), and they have the same split-structure. After creation of split-joint-arc (Algo. 7, Line 6), the nodes leave the split structure (Algo. 7, line 7, line 16). The concatenation of parallel structures is possible, when paths of a same split-structure do not meet in a same split-joint structure. The node ending split $s$ is then included in the parallel structure split (Algo. 7, line 11).

*Algorithm 8:* CheckSplit (Service $s$, Temporary buffer $T$)

```
1: for i = 1, |T| − 1 do
2:    for j = i + 1, |T| do
3:       if (M(Out_{T[i]}, Out_{T[j]}) > ε) then
4:          AddArcSplit(s, T[i], T[j]);
5:          GetIntoSameSplit(T[i], T[j]);
6:       end if
7:    end for
8: end for
```

The checking for the existence of a split structure (Algo. 8) is performed between the current node and the nodes in the temporary buffer $T[j]$. If these nodes have different functionalities (i.e., different output), we create a split-arc and add them in the same split structure (Algo. 8, line 5). Then all following nodes are affected to this split structures (Alg. 2, line 15-16). When all nodes of the current layer are processed, the next layer becomes the current layer and so on until the next layer is empty. The algorithm terminates when this state is reached.

### C. Algorithm Complexity

An algorithm complexity can be evaluated from two aspects. Time complexity measures the processing time, while Memory complexity measures the manipulated data size.

*1) Time Complexity:* In the algorithm 2, each request functionality ($\mathcal{F}_{\mathcal{R}}[i]$) implies at least one iteration of the algorithm. This leads to a total of at least $card(\mathcal{F}_{\mathcal{R}})$ iterations. An iteration involves $\bar{L}$ ($\bar{L}$ is the average services layers size) discovered services, and for each discovered service there are $card(\mathcal{D})$ verifications of condition, $card(C_s)$ verifications of service constraint , and $O(split - joint)$ verifications of split-joint if the discovered service belongs to a split structure. For each iteration, the algorithm checks

$O(split)$ times the split structure. Discovered informative services are executed and all their provided data verified.

The algorithm has then a complexity order of $card(\mathcal{F}_\mathcal{R}) \times \bar{L} \times card(\mathcal{D}) \times card(C_s) \times O(split) \times O(split-joint)$. Condition and constraint checking have a constant complexity, but the split

checking process has a complexity of *vector sort*, in the worst case $O(split) = \bar{L} \times log(\bar{L})$. The split-joint checking process has a complexity order $\bar{L}$. In conclusion, in the worst case, the algorithm complexity equals $O(card(\mathcal{F}_\mathcal{R}) \times \bar{L} \times (\bar{L}^2 \times log(\bar{L})) \times card(\mathcal{D}) \times card(C_s))$. Since the discovered service $\bar{L}$, are important, then the complexity algorithm is cubic and has an order $\bar{L}^3$.

The complexity of algorithm 1 equals $O(card(\mathcal{F}_\mathcal{R}) \times card(C_s) \times \bar{L})$. This complexity is linear because $card(\mathcal{F}_\mathcal{R}) \times card(C_s)$ is constant.

To conclude, our algorithms have a cubic complexity (Algo. 2) or linear complexity (Algo. 1). The most influential element is the number of discovered services. Therefore, the more services answering the request functionality, the more important is the resolution time. The number and severity of constraints put on services and data in the request will reduce the number of services. However too much or strict constraints can lead to empty layers in the composition graph.

*2) Memory Complexity:* It is important to reduce the memory used by the composition graph. Generally, we can store the graph nodes and their arcs, but we cannot save all paths of the graph, because the memory complexity is exponential $(\bar{L}^l)$, where $\bar{L}$ is the average layer size and $l = card(\mathcal{F}_\mathcal{R})$ is the layer number.

Concerning ACG, the nodes size is: $O(card(N)) = r + (l \times (2 \times \bar{L})) + 2 \approx l\bar{L}$, where $r = card(\mathcal{D})$ is the condition number. We multiply $\bar{L}$ by 2, because each functionality can require both informative and active services, and $(+2)$ corresponds to the start and the end node $(sn, en)$. The arcs size is: $O(card(A)) = l \times (\bar{L}^2 + \bar{L} \times r) \approx l \times \bar{L}^2$,

Concerning MSCG, in the worst case, the nodes size is: $O(card(N)) = l \times r \times (\bar{L} + \bar{L}) + 2 = 2lr\bar{L}$. The arcs size is: $O(card(A)) = l \times (r \times \bar{L}^2) + r \times (\bar{L} + \bar{L}) = l \times r \times \bar{L}^2 + 2 \times l \times r \times \bar{L}$, which can be approximated by: $O(card(A)) \approx l \times r \times L^2$. However, the memory complexity of a transformed MSCG is the same as for ACG.

## VII. Experimentation

The experiments we have conducted focus on three points: complex request description and services composition automatic design. For the request description, we give some example requests in natural language, and then describe manually their functionalities, conditions, constraints and objectives. For the automatic composition, we check its correctness by assessing the suitability of composition structures regarding the request and available services. The second point is the processing time of the composition design algorithm. Based on its complexity, the size of matching services has an important impact for the processing time.

### A. Implementation and Hypotheses

The used Web services are described semantically, using the OWL-S Language. For that purpose, we implemented a program generating OWL-S descriptions of services. This program relies on the Jdom API [42]. The type of service (input and output) is defined by referring to a domain ontology of travel booking The values of QoS parameters are randomly generated: price and reputation of service are considered.

We used different APIs at the different levels of the resolution process. (1) Jena [43], and SPARQL [44] are used to check the data constraints ($C_d$) and conditions ($\mathcal{D}$). (2) The OWL-S API [45] is used to read the OWL-S service description and also to check services constraints ($C_s$). (3) Pellet [46] is used to check the matching level between $I/O$ of services and the request, through the OWLS-MX API.

Our experiments have been conducted with the following set of hypotheses:

(1) Web services are formally described with OWL-S. We do not consider mapping with other Web services semantic description language.

(2) A QoS ontology is used within the OWL-S services descriptions, in order to describe the non-functional parameters such as price and reputation.

(3) A specific service domain ontology linked to a domain ontology is used to describe service I/O. In our experimentation cases, we have used the Travel Booking ontology which is associated to the MobileTTE Tourism ontology [47]. Request and services use this same ontology.

(4) We do not consider the necessary mapping of services described with different domain ontologies: all the services are described using a same travel booking ontology.

(5) As a consequence of (4), the matching process provides binary responses, even if the OWLS-MX API [12] that we use implements a full matchmaking process. Moreover only inputs and outputs parameters are exploited in the matchmaking.

Finally, the computing environment used to implement and test the application is a Core2 Duo CPU based machine (1.58 GHz) with 2.89 GB of RAM.

### B. Complex Request Description

The tourism domain has been chosen as use case, because it offers a rich set of services. Moreover, the planning of a tourism trip is a classical example of service composition problem that anyone can face today. We consider the following request: *"We want to travel from City A to City B, to reserve several individual hotel rooms in destination city where each booking is billed separately, rent a car for six people and if a car does not allow six people, then rent two cars. I want to know the weather forecast and get a map*

*of the destination city. Additionally, choose a museum visit or city monument visit according to the weather. The whole at best price and reputation, and the price must not exceed 3000 Euro"*.

This request has four functionalities: transport, hotel, car renting, city information. Each request functionality is characterized with the input and output class. Theses classes are detailed in Table III.Note that the city information functionality can be answered with different services such as, city weather, city map, and city museum service. These services can be executed in parallel because they have the same input type and provides different output types (as detailed in Table I).

This request contains three conditions, $\mathcal{D} = \{d_1, d_2, d_3\}$ :

(1) A condition referring to the hotel billing ($d_1$): *"each booking is billed separately"*. This condition is written as, $d_1 : if\ (a_1 < a_2) \Rightarrow c_1 = false$, where $a_1$ is the reserved rooms number, $a_2$ is the billed reservations and $c_1$ is the achieved payment.

(2) A condition referring to the weather state ($d_2$): *"choose the museum visit or city monument according to the weather"*. This condition is written as, $d_2 : if\ (a_3 = good) \Rightarrow c_2 = c_3$. Where $a_3$ is the weather condition, $c_2$ is cultural activity and $c_3$ is monument visit.

(3) A condition referring to the rental car type ($d_3$): *"when a car does not allow six people, then rent two cars"*. This condition is written as, $d_3 : if\ (a_4 < 6) \Rightarrow c_4 = 2$. Where $a_4$ is the car capacity and $c_4$ is the number of cars.

The request predicates $a_1, a_2, a_3, a_4, c_1, c_2, c_3, c_4$, correspond to properties in the domain ontology. $a_1$ and $a_2$ are properties of the "`BookedHotelInput`" class. $a_1$ is the data property "`numberOfrooms`", $a_2$ is the object property "`paymentBookHotel`". $a_3$ is a property of the "`WeatherOutput`" class, and corresponds to the data property "`weather-condition`". $a_4$ corresponds to the data property "`carCapacity`", and it is a property ofthe "`CarInfo`" class, and so one.

The request contains two objectives, price and reputation, and one constraint, the price. The price and reputation are described in the QoS ontology. Formally the request elements are: $I_{\mathcal{R}} = \{I_1, I_2, I_3, I_4\}$, $O_{\mathcal{R}} = \{O_1, O_2, O_3, O_4\}$, $\mathcal{D} = \{d_1, d_2, d_3\}$, $\mathcal{C} = \{C_d, C_s, C_c\}$, $C_d = \phi$, $C_s = \phi$, $C_c = \{price\}$, $\mathcal{B} = \{B_d, B_s, B_c\}$, $B_c = \{price, reputation\}$.

Table IV details some experimented requests. The initial request, described at the beginning of this section can be formalized with $\mathcal{R}_2$ or $\mathcal{R}_4$, where the request functionalities order is different. The request $\mathcal{R}_1$ is less complex and corresponds to: *"I want to travel from City A to City B, reserve an hotel room in destination city and rent a car. The whole at best price and reputation, and the price does not exceed 3000 Euro"*.

The request $\mathcal{R}_3$ corresponds to request $\mathcal{R}_1$ by adding the requirements: *"(1) reserve several hotel rooms and each booking is billed separately; (2) rent a car for six people and when a car does not allow six people, then rent two cars; (3) know the weather forecast and get a map for the destination city"*.

We choose these specific requests, in order to evaluate the correctness of the automatic composition, by analyzing the different compositions designed by the algorithm. The requests $\mathcal{R}_1, \cdots, \mathcal{R}_4$ are described semantically with the OWL-CR ontology.

In the following section, we present our experiments and the obtained results. The experiments rely on different use cases, each use-case corresponding to a specific request.

### C. Automatic Composition Results

Algorithm, 1 and 2 have been used to answer the request. In our tests, the matching threshold has fixed to 1, and the similarity threshold to 0.

The composition design algorithms search the available services in a UDDI, in order to design answer(s) for a given request. This UDDI contained references of services covering in particular the functionalities of the request: hotel, transport, rent a car and city information.

In order to evaluate the algorithm performances, we present and discuss two types of experiments. The first one concerns the correctness of composition. The second one concerns the composition performances in terms of response time.

*1) Composition Correctness:* The assessment of the composition correctness consists in verifying that the designed composition answers all request functionalities and detects correctly the composition structures whatever their numbers and the number of different ones.

Figures 5 and 6 illustrate a composition graph answering the request $\mathcal{R}_1$, where the rectangle presents the *informative service*, a circle presents an *active service* (the same meaning for the following Figures), and the graph contains only the choice and sequence structures. The graph on Figure 5 contains 10 answering services by layer. For better visibility, the graph on Figure 6 contains only 2 answering composition by layers.

Observing the graph, it can be seen that, the transport request is answered either by an atomic service, $Booktransport$, or a composite service $\{Availabletransport \rightarrow BookFlight\}$. This illustrates that, for each request functionality, the algorithm can find one single matching service or builds sub-composition.

The Figure 7 illustrates the composition graph resulting from the processing of request $\mathcal{R}_2$, which requires multiple composition structures: sequence, while, choice, switch, split and split-joint. The graph illustrates clearly the composition structures and for more visibility, each layer is limited to two services. We recall that arcs $\rightarrow, \vdash, \dashv$ illustrate respectively: sequence, split, and split-joint. The structure switch "$sw_2$"

| Class | Properties |
|---|---|
| $I_1 : InTransportClass$ | $departCity$, $destinationCity$, $travelDate$ |
| $I_2 : InHotelClass$ | $roomNumber$, $arrivedDate$, $departDate$ |
| $I_3 : InRentCarClass$ | $dateTime4Take$, $dateTime4Make$, $InRentCarClass$ |
| $I_4 : InCityInformation$ | $cityName$, $Date$ |
| $O_1 : OutTransportClass$ | $flightOrTrainNumber$ , $reservationNumber$, $placeNumber$ |
| $O_2 : OutHotelClass$ | $reservationNumber$ |
| $O_3 : OutRentCarClass$ | $reservationNumber$ |
| $O_4 : OutCityInformation$ | $Map$, $weather$, $Heritage$, $Museum$ |

Table III
INPUT AND OUTPUT CLASSES

| Request | Formalization |
|---|---|
| $\mathcal{R}_1$ | $< \{I_1, I_2, I_3\}, \{O_1, O_2, O_3\}, \mathcal{D} = \phi, \mathcal{C} = \{price\}, \mathcal{B} = \{price, reputation\}, \lambda = \phi >$ |
| $\mathcal{R}_2$ | $< \{I_1, I_2, I_3, I_4\}, \{O_1, O_2, O_3, O_4\}, \{d_1, d_2, d_3\}, \{price\}, \{price, reputation\}, \phi >$ |
| $\mathcal{R}_3$ | $< \{I_2, I_1, I_3, I_4\}, \{O_2, O_1, O_3, O_4\}, \{d_1, d_3\}, \{price\}, \{price, reputation\}, \phi >$ |
| $\mathcal{R}_4$ | $< \{I_2, I_1, I_4, I_3\}, \{O_2, O_1, O_4, O_3\}, \{d_1, d_2, d_3\}, \{price\}, \{price, reputation\}, \phi >$ |

Table IV
EXPEREMENTED REQUEST

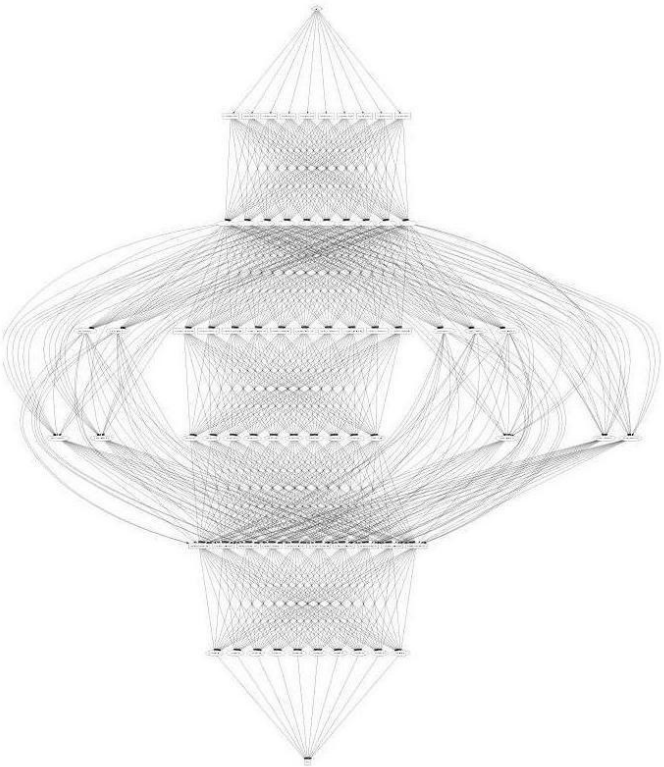

Figure 5. Executable service composition graph for $\mathcal{R}_1$ (10 services)

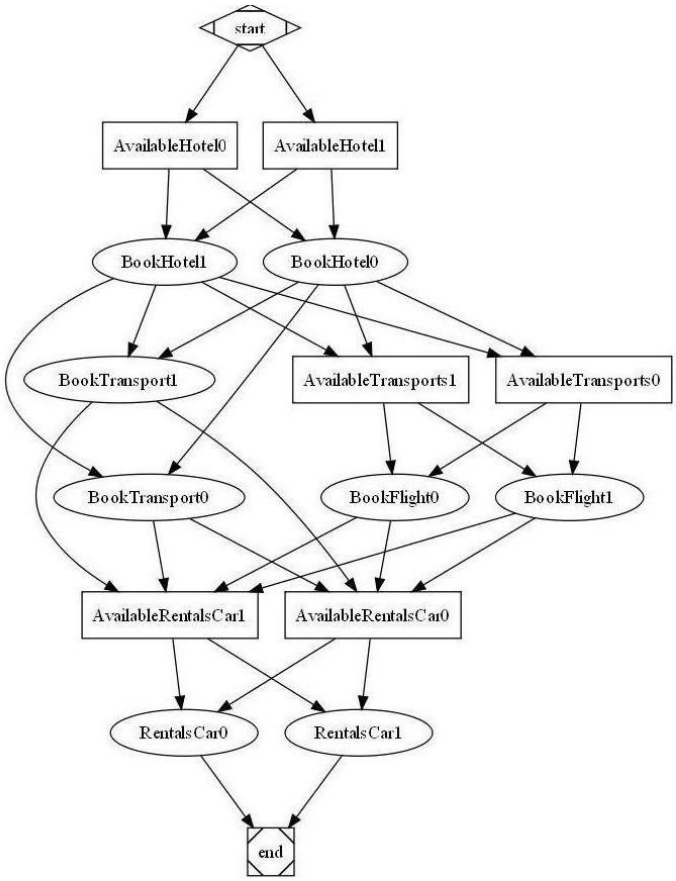


Figure 6. Executable service composition graph for $\mathcal{R}_1$ (2 services)

checks the city weather and it belongs to the parallel structure (between split and split-joint), this shows that the automatic composition algorithm can provide an imbrication of composition structures.

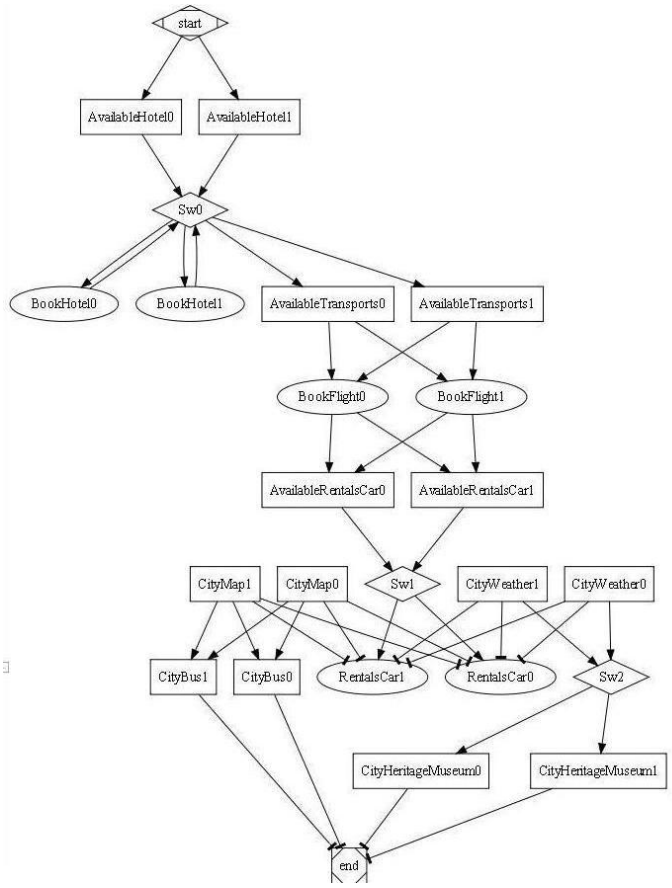


Figure 7. Executable service composition graphs for $\mathcal{R}_2$

The graph composition structures, illustrated Table V, are coherent with structure characteristics defined in Table I. For example, AF and AT are structured with a "choice" structure because their output classes match. AT and BF are structured with a "sequence" because the output of AT matches with the input of BF. BH is structured with a "while" because "each booking is billed separately"; the condition checking is before and after BH. A "switch" structure sw1 follows ARC service because the request condition "*rent a car for six people*" refers to the car capacity predicate and it belong to the output of ARC. RC is followed by structure a "split", opening parallelism, because the following services CW and CM have different functionalities. CB and CHM are followed by a "split-joint", ending parallelism, because they are included in the same split structure and are followed by the same node "end". The same holds for the other structures (Figure 8 and Figure 9).

In the built composition graphs, we have tested if the composition structures can be detected correctly. After this

| Composition structure | participating services |
|---|---|
| Choice | (AH0,AH1), (BF0, BF1), (AH0, AH1), etc |
| Sequence | (AT0, BF0),(AT0, BF1),(BF0, AH0), etc |
| While | BH0, BH1 |
| Switch | sw0, sw1, sw2 |
| Split | RC0, RC1 |
| Split-joint | end |

Table V
STRUCTURES IN THE COMPUTED COMPOSITIONS,

where, AT denotes Available Transport, AF denotes Available Flight, BT denotes Book Train, BF denotes Book Flight, AH denotes Available Hotel, BH denotes Book Hotel, ARC denotes Available Rentals Car, RC denotes Rent Car , CB denotes City Bus, CW denotes City Weather, CM denotes City Map, and CHM denotes City Heritage Museum.

verification, we test the limit of this detection, such as, (1) test if the services following split structure can proceed, at the same time, the split-joint structure. (2) test if a parallel structure, split and split-joint can be in the middle of a composition path.

The Figure 8 and Figure 9 illustrate the composition graph resulting from the processing of request $\mathcal{R}_3$, and $\mathcal{R}_4$ respectively. The view is limited to two services by layers for sake of readability.

The graph on Figure 8 illustrates the services following a split and preceding a split-joint in the same time. In this test we have removed from the UDDI registry, the following services: ''City Heritage Museum'' and ''City bus''.

The graph on Figure 9 illustrates the possibility that the split and split-joint can be detected in the middle of the composition. The parallel structure follows BookFlight, and precedes AvailabeRantelsCar.

*2) Performance Experiment:* As said in section VI-C1, the complexity of composition algorithms depend strongly on the answering services number (the width of its layers). We have conducted experiments to evaluate this influence.

Considering the composition graph answering the request having four request functionalities ($\mathcal{R}_2$), the Table VI summarizes the layer size impact on the composition design response time. Where $card(N)$, $card(A)$ are respectively nodes and arcs size of the composition graph, $\bar{L}$ is the average layer size. We recall from the analysis made in section VI-C2, that $card(N) \approx 2lr\bar{L}$ and $card(A) \approx lr\bar{L}^2$, where $r$ is the number of conditions of the request and $l$ the number of layer. The Figure 10 illustrates the evolution of the execution time with respect to the average layer size.

If we consider that averagely each functionality can have about 15 answering services, the request processing requires $600\ seconds$ (about $10\ minutes$) to return the MSCG and about $3\ minutes$ to return ACG. As shown in Table VI and VII, the algorithm processing time depends on the number of answering services ($\bar{L}$) can be limited thanks to request constraints. We note that the larger the number of request constraints (services $C_s$ and data $C_d$), the smaller

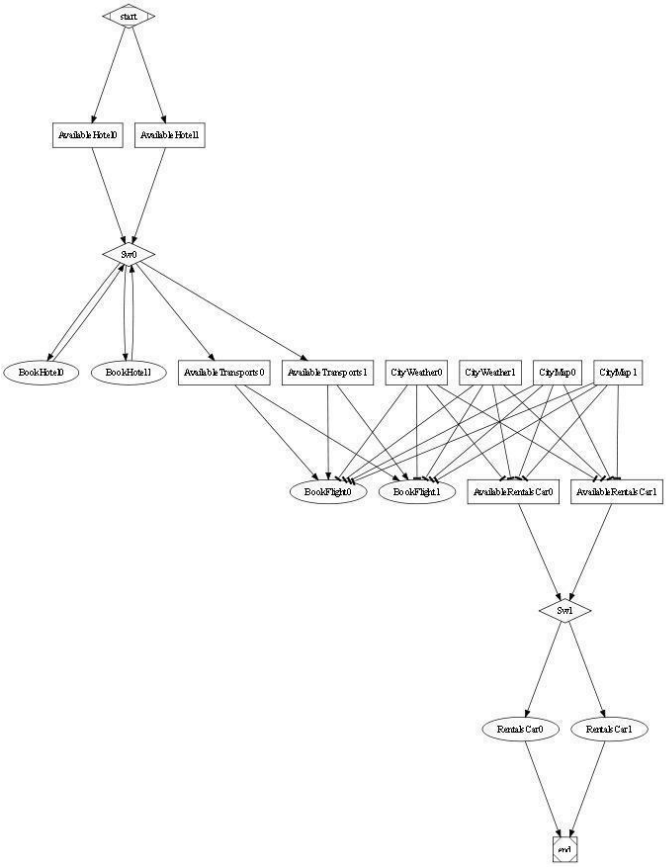


Figure 8. Executable service composition graph for $\mathcal{R}_3$

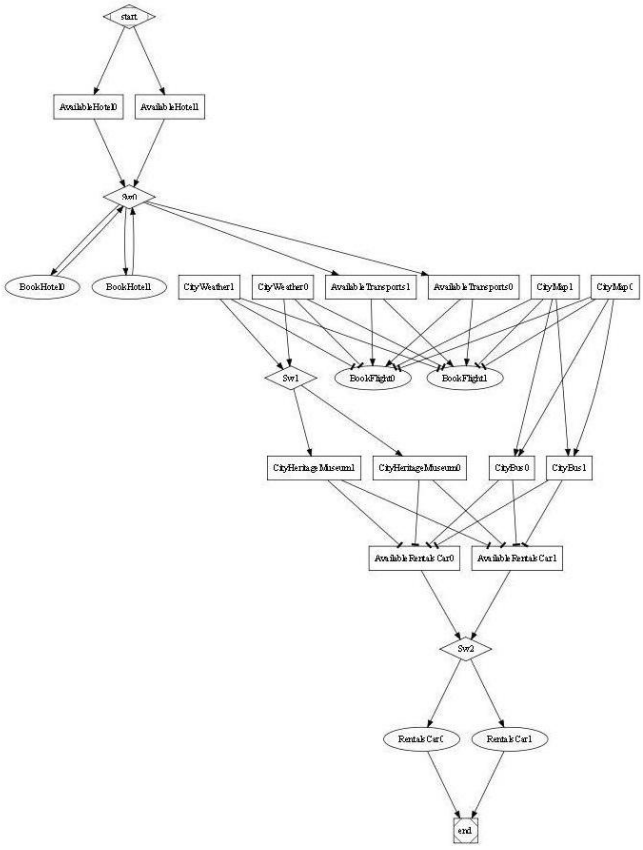


Figure 9. Executable service composition graph for $\mathcal{R}_4$

| $\bar{L}$ | $card(N)$ | $card(A)$ | time (second) |
|---|---|---|---|
| 2 | 24 | 42 | 25 |
| 4 | 45 | 124 | 57 |
| 6 | 65 | 246 | 89 |
| 8 | 85 | 408 | 108 |
| 10 | 105 | 610 | 194 |
| 12 | 125 | 852 | 333 |
| 14 | 145 | 1134 | 484 |
| 16 | 165 | 1456 | 740 |
| 18 | 185 | 1818 | 1098 |
| 20 | 205 | 2220 | 1641 |

Table VI
DISCOVERED SERVICES INFLUENCE ON RUNTIME WITH MSCG.

| $\bar{L}$ | $card(N)$ | $card(A)$ | time (second) |
|---|---|---|---|
| 2 | 16 | 32 | 28 |
| 4 | 30 | 120 | 49 |
| 6 | 44 | 264 | 73 |
| 8 | 58 | 464 | 94 |
| 10 | 72 | 720 | 119 |
| 12 | 86 | 1032 | 143 |
| 14 | 100 | 1400 | 171 |
| 16 | 114 | 1824 | 194 |
| 18 | 128 | 2304 | 220 |
| 20 | 142 | 2840 | 248 |

Table VII
DISCOVERED SERVICES INFLUENCE ON RUNTIME WITH ACG.

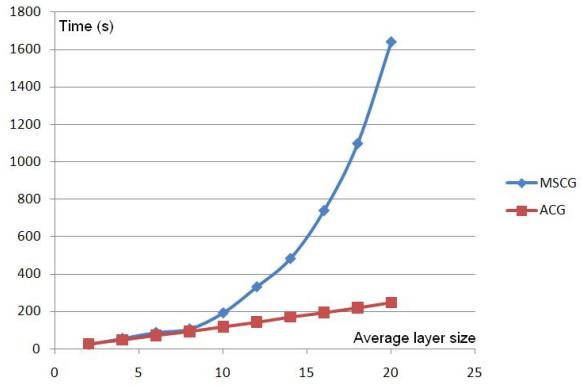


Figure 10. Dependence between average layer size and execution time.

the number of answering services and then the smaller the processing time.

## VIII. SPECIFIC CASES

The complex request resolution method changes according to the characteristics of composition graph, request dependencies type and dynamic events. In this section, we discuss these specific cases.

### A. One Step and Multi Steps Resolution

The composition graph can be a layered digraph (order pair of sets, as illustrated in Figure 3) in which the nodes of too consecutive layers are all connected together, (i.e., there exists an arc $a_{u,v}$ between each node $u$ and $v$ of consecutive graph layers, $l_k$ and $l_{k+1}$). This case is formalized by the following condition:

$$\forall i \in \{1 \cdots k\}, \forall u \in l_i, \forall v \in l_{i+1} : a_{u,v} \in A. \quad (7)$$

At design time, the selection of compositions answering a request may or may not require an optimization method, depending on the composition graph structure and the number of objectives in the request. Considering the request functionalities have no global dependency, the composition does not require an optimization method when: (a) the graph layers respect the condition 7 and there is only a request objective ($card(\mathcal{B}) = 1$); or (b) the objectives can be combined because their importance levels $\lambda$ are fixed ($\lambda_i = x, x \in [0, 1]$), thus allowing to order them. Optimization is not required when condition 7 is verified and the following condition is verified:

$$(card(\mathcal{B}) = 1) \vee (card(\mathcal{B}) > 1 \wedge \lambda \neq \phi \wedge \lambda_i = x) \quad (8)$$

In this case, the request functionalities can be treated separately, for each request functionality, and several matching services can be found. The more suitable one is selected according to the request objectives and constraints, and then executed. In other words, if this condition is verified, the steps: (2) service discovery, (3) composition design, (4) optimization of compositions, (5) execution of compositions, (illustrated in Figure 1) can be merged in one step.

In any other case, an optimization method is required to select the best composition. Therefore, we distinguish two types of request processing: one step resolution (no optimization) and multi-steps resolution (with optimization), as illustrated in figure 11.

The automatic composition is also adapted consequently: composition with selection for one step resolution, and composition without selection for multi-steps resolution. The compositions selection without optimization method means that this selection is done during the composition (left-hand side of Figure 12). When the composition selection is done with an optimization method, the composition design provides the set of composed services (right-hand side of Figure 12), in order to use it as an optimization search space.

The composition design steps are finally the following, the required steps depending on the need for optimization:

(0) Formalize user request with OWL-CR. (1) Read the request functionalities. Then for each functionality: (2) Find the corresponding services using the `discover` process, which uses the matchmaker process for computing a matching level between request input/output and existing services. The discovered services and data are then filtered, according to the service constraints ($c_s$) and data constraints ($c_d$).

(3) The `Composition design` takes into account the possible composition structures by considering the request conditions $\mathcal{D}$.

(4) Select the best service according to the request objective(s) $\mathcal{B}$.

(4A) and (5A): Go to the next functionality, if the discovered (4A)/selected (5A) service fulfills the current functionality.

(4B) and (5B): consider again the same treated functionality, if the discovered (4B) or selected (5B) service does not fulfill the functionality. In this case, we consider the discovered or selected service output as a next request input.

### B. Global Dependencies and Compositions Model

The existence of global dependencies generates some complexity in composition modeling. This is because the compositions graph will be designed with multiple layers of matching services and only subsequent layers can have relation arcs between them. Indeed, the checking at each node of the arcs history requires to maintain a list of all preceding paths to consider the relations with the following layers, which is a task having an exponential complexity [48]. The memory complexity for a node in the graph can be $\bar{L}^l$, where $\bar{L}$ is the average number of layers and $l$ is the number of layer. Let $n_i, n_{i+1}$ be nodes; to check the existence of an arc $a_j : n_i \rightarrow n_{i+1}$, we may require to check conditions related to nodes $[n_1 \cdots n_{i-1}]$.

We can neglect the path history by modeling the set of compositions as a graph, but without any guaranty regarding the validity of the composition. To be able to considering valid compositions only, we model the compositions as a set of clusters, each cluster corresponds to a set of services answering a request functionality. Then the validity of the compositions is checked during the initial step of the optimization.

### C. Dynamic Events Processing

As said in section I-A, the steps (3-5: composition, optimization, execution) can be concerned by dynamical events, because on the one hand, random events can lead to service breakdown, affecting quality of service, and on the other hand because new services can appear. This impacts the composition design, the composition optimization and the composition execution.

During the composition design, perturbations can be neglected, because the composition design takes few time, and considers available services only.

During composition optimization, a specific mechanism has to be taken into account.

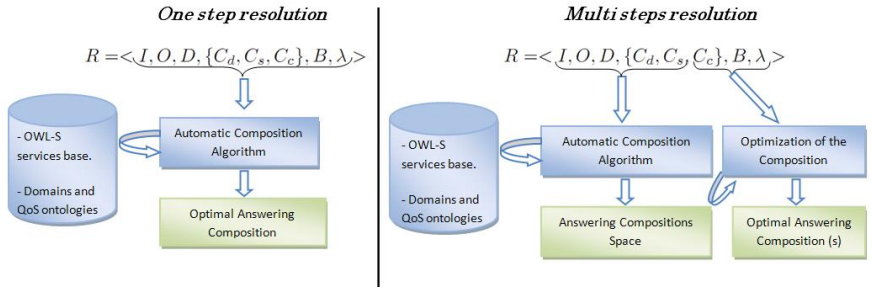


Figure 11. Solving request steps

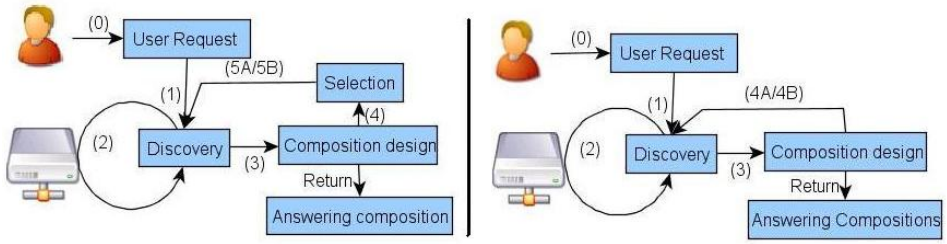


Figure 12. Automatic composition flow

During composition execution, the events are considered within the composition representation. When this one does not provide an alternative solution, we create a sub-composition corresponding to a new request and then select the best sub-composition with or without optimization. The new created request considers only the functionalities, that have not been fulfilled. We consider a functionality is completely achieved only when the corresponding active service is executed. Let $i$ be an index of the last completed functionality, the new request is formalized as:

$$\mathcal{R} =< \mathcal{F}_{[i+1,...]}, \mathcal{D}, \mathcal{C}, \mathcal{B}, \lambda > \quad (9)$$

Then we reuse an automatic composition algorithm (see section VI). In order to reduce the time of resolution, we suggest using algorithm 1 instead of algorithm 2, because it has lower resolution complexity.

To estimate the possibility of reparation, without recreating sub-composition that already exit in the graph, we calculate the $k-$connectivity of the composition graph.

The $k-$connectivity of a composition graph evaluates how many nodes can breakdown with assuring that the graph has the compositions alternatives. For example, if k=3, then for each depart-destination node, maximally two nodes can be deleted for assuring the possibility of finding an alternative path ($k-1$: maximal number of nodes, whose can be deleted). For a fully connected composition graph, $k$ corresponds to a minimal size of existing layers, $k = min(card(L))$.

## IX. Conclusion and Future Work

This paper proposes a groundwork for solving complex Web requests with Web services composition considering the request processing steps. The set of compositions are modeled according to the request characteristics ( request objectives and constraints). When there exist global dependencies, the set of possible compositions can not be modeled as a graph. In this case, compositions are modeled as a set of clusters. When the request dependency is not global, the set of compositions is modeled as a directed graph. The latter can be built with the algorithms we proposed.

Our algorithms answer a request, and in order to define answering compositions, exploit existing Web services and a semantic description of both request and services. If the request objectives or constraints have different evaluation function for different composition structures, then the composition algorithm 2 can be used. Otherwise, algorithm 1 is used. The composition graphs built by the algorithms, MSCG or ACG, can be used as a search space for the optimization process. In addition, if one of the composed services breaks down, the proposed algorithms support the repairing of the original composition.

We cited conditions allowing to choose automatically the suitable composition algorithm, suitable model of compositions and suitable resolution steps.

Experiments show the correctness of obtained compositions, concerning all composition structures, and the runtime performance. When only sequence structures are considered, the runtime complexity is linear with the average Web services. When all composition structures are considered,

the runtime complexity is cubic.

In the future work, we look: (1) to consider better the personalized knowledge of user, [49], [50] consider the composition answering a request as a personalized composition. According to our point of view, the answering composition is personalized when it answers a request by considering its conditions $\mathcal{D}$. The personalization can be widen, by considering the user profile for example. (2) To define the process allowing the transformation, from MSCA, ACG or clusters set to OWL-S or BPEL4WS description. This transformation does not consider the data (input values if services) and concerns only the non-personalized composition, in order to add them to the services base.

## ACKNOWLEDGMENT

This work has been performed under the PhD grant TR-PhD-BFR07 funded by the Luxembourg National Research Fund (FNR).

## REFERENCES

[1] C. H. Marcussen, "Trends in European Internet Distribution - of Travel and Tourism Services," Tech. Rep., 2009. [Online]. Available: http://www.crt.dk/uk/staff/chm/trends.htm

[2] B. Batouche, Y. Naudet, and F. Guinand, "Algorithm to solve web service complex request using automatic composition of semantic web service," in *COGNITIVE*, 2010, pp. 84 – 89.

[3] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, 2004, pp. 43–54.

[4] D. B. CLARO, " SPOC - Un canevas pour la composition automatique de services web ddis la ralisation de devis," Ph.D. dissertation, Universit d'angers, october 2006.

[5] B. Jeong, H. Cho, and C. Lee, "On the functional quality of service (fqos) to discover and compose interoperable web services," *Expert Syst. Appl.*, vol. 36, pp. 5411–5418, April 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1497653.1498400

[6] Z. Li, L. O'Brien, J. Keung, and X. Xu, "Effort-oriented classification matrix of web service composition," in *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, ser. ICIW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 357–362. [Online]. Available: http://dx.doi.org/10.1109/ICIW.2010.59

[7] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with owl-s," *World Wide Web*, vol. 10, pp. 243–277, September 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1285732.1285745

[8] C. REY, "D 2 CP et computeBCov: Un prototype et un algorithme pour la découverte de services web dans le contexte du web sémantique," *Ingénierie des systèmes d'information(2001)*, vol. 8, no. 4, pp. 83–112, 2003.

[9] M. Klusch, B. Fries, and K. Sycara, "Automated semantic web service discovery with owls-mx," in *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM Press, 2006, pp. 915–922. [Online]. Available: http://dx.doi.org/10.1145/1160633.1160796

[10] U. Rerrer-Brusch, "Service Matching with Contextualised Ontologies," Ph.D. dissertation, University of Paderborn, october 2006.

[11] D. Martin, M. Burstein, E. Hobbs, O. Lassila, D. Mcdermott, S. Mcilraith, S. Narayanan, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "OWL-S: Semantic Markup for Web Services," Nov. 2004. [Online]. Available: http://www.w3.org/Submission/OWL-S/

[12] M. Klusch, B. Fries, and M. Khalid, "Owls-mx: Hybrid owl-s service matchmaking," in *Proceedings of 1st Intl. AAAI Fall Symposium on Agents and the Semantic Web*, 2005.

[13] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," in *WWW*, 2002, pp. 77–88.

[14] R. Hull and J. Su, "Tools for composite web services: a short overview," *ACM SIGMOD Record*, vol. 34, no. 2, pp. 86–95, 2005.

[15] A. Brogi and S. Corfini, "Behaviour-aware discovery of web service compositions," International journal of Web services research, Tech. Rep., 2006.

[16] P. Xiong, Y. Fan, and M. Zhou, "Qos-aware web service configuration," *IEEE Transactions on Systems Man and Cybernetics Part A Systems and Humans*, vol. 38, no. 4, pp. 888–895, 2008.

[17] X. Tang, C. Jiang, and M. Zhou, "Automatic web service composition based on horn clauses and petri nets," *Expert Syst. Appl.*, vol. 38, pp. 13 024–13 031, September 2011. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2011.04.102

[18] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng, "Quality driven web services composition," *Proceedings of the 12th international conference on World Wide Web*, pp. 411–421, 2003.

[19] X. X. Yifei Wang, Hongbing Wang, "Web Services Selection and Composition based on the Routing Algorithm," *10th IEEE International Enterprise Distributed Object Computing Conference Workshops*, pp. 57–66, 2006.

[20] J. Cardoso and A. Sheth, "Semantic e-workflow composition," *Journal of Intelligent Information Systems*, vol. 21, pp. 191–225, 2003.

[21] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus," pp. 159–178, 1998.

[22] J. T. E. Timm, "Specifying semantic web service compositions using uml and ocl," in *In 5th International Conference on Web Services*. IEEE press, 2007.

[23] R. Kazhamiakin and M. Pistore, "A parametric communication model for the verification of bpel4ws compositions," in *In Mario Bravetti, Lela Kloul, and Gianluigi Zavattaro, editors, EPEW/WS-FM, volume 3670 of Lecture Notes in Computer Science*. Springer, 2005, pp. 318–332.

[24] D. Pellier and H. Fiorino, "Un modle de composition automatique et distribue de services web par planification," in *Revue d'Intelligence Artificielle, volume 23*, no. 1, 2009, pp. 13–46.

[25] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for web service composition," in *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[26] S.-C. Oh, B.-W. On, E. J. Larson, and D. Lee, "Bf*: Web services discovery and composition as graph search problem," *e-Technology, e-Commerce, and e-Services, IEEE International Conference on*, pp. 784–786, 2005.

[27] M. Klusch and A. Gerber, "Semantic web service composition planning with owls-xplan," in *In Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, 2005, pp. 55–62.

[28] E. M. Goncalves da Silva, L. Ferreira Pires, and M. J. van Sinderen, "An algorithm for automatic service composition," in *1st International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, ICSOFT 2007, Barcelona, Spain*, E. M. Goncalves da Silva, L. Ferreira Pires, and M. J. van Sinderen, Eds. INSTICC Press, July 2007, pp. 65–74.

[29] S.-C. Oh, D. Lee, and S. R. T. Kumara, "Web service planner (wspr): An effective and scalable web service composition algorithm," *Int. J. Web Service Res.*, vol. 4, no. 1, pp. 1–22, 2007.

[30] C. Rey, "Dcouverte des meilleures couvertures d'un concept en utilisant une terminologie Application la dcouverte de services web smantiques," Ph.D. dissertation, Universit Blaise Pascal - Clermont II, dcembre 2004.

[31] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, "Web services. concepts, architectures and applications," 2003.

[32] UDDI, "Uddi technical white paper," September 2000.

[33] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web services architecture," World Wide Web Consortium, Note NOTE-ws-arch-20040211, February 2004.

[34] L. Bourgois, "Représentation et comparaison de Web services complexes avec des logiques dynamiques," Ph.D. dissertation, Universite Paris 13- Villetaneuse, june 2007.

[35] J. Schaffner, H. Meyer, and M. Weske, "A formal model for mixed initiative service composition," *Services Computing, IEEE International Conference on*, pp. 443–450, 2007.

[36] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, *BPEL4WS, Business Process Execution Language for Web Services Version 1.1*, IBM, 2003. [Online]. Available: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[37] S. V. Hashemian and F. Mavaddat, "Automatic composition of stateless components: a logical reasoning approach," in *Proceedings of the 2007 international conference on Fundamentals of software engineering*, ser. FSEN'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 175–190. [Online]. Available: http://portal.acm.org/citation.cfm?id=1775223.1775235

[38] L. H. P. G. Dan, A., "Web service differentiation with service level agreements," White Paper, IBM Corporation, Tech. Rep., March, 2003.

[39] "http://www.w3.org/Submission/WSDL-S/," 12-12-2011.

[40] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Dynamic Web Service Composition in METEOR-S," 2004.

[41] J. Cardoso, J. Miller, A. Sheth, and J. Arnold, "Modeling Quality of Service for Workflows and Web Service Processes," *Web Semantics Journal: Science, Services and Agents on the World Wide Web Journal*, vol. 1, no. 3, pp. 281–308, 2004.

[42] "http://www.jdom.org/docs/apidocs/," 12-12-2011.

[43] hp, "Jena - A Semantic Web Framework for Java," available: http://jena.sourceforge.net/index.html, 2002.

[44] E. P.hommeaux and A. Seaborne, "Sparql query language for rdf (working draft)," W3C, Tech. Rep., March 2007. [Online]. Available: http://www.w3.org/TR/2007/WD-rdf-sparql-query-20070326/

[45] "http://www.mindswap.org/2004/owl-s/api/," 12-12-2011.

[46] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.websem.2007.03.004

[47] J.-D. Labails, "Description de l'ontologie des ressources touristiques," MobileTTE WP5.1: Standard d'interoprabilit pour les donnes touristiques, Technical Report, Henri Tudor Public Research Center, Luxembourg, Tech. Rep., 2006.

[48] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[49] P. Albers and O. Licchelli, "Composition de services web personalis," in *Intelligence Artificielle et Web Intelligence Conference, Grenoble*, July 2007.

[50] S. Khapre and D. Chandramohan, "Personalized web service selection," in *International Journal of Web Semantic Technology (IJWesT) Vol.2, No.2*, April 2011.