Scalable Computing: Practice and Experience
Volume 12, Number 1, pp. 137–150. http://www.scpe.org
ISSN 1895-1767

ANT-INSPIRED FRAMEWORK FOR AUTOMATIC WEB SERVICE COMPOSITION

CRISTINA BIANCA POP, VIORICA ROZINA CHIFU, IOAN SALOMIE, MIHAELA DINSOREANU, TUDOR DAVID, VLAD ACRETOAIE*

Abstract. This paper presents a framework for automatic service composition which combines a composition graph model with an Ant Colony Optimization metaheuristic to find the optimal composition solution. The composition graph model encodes all the possible composition solutions that satisfy a user request. The graph will be further used as the search space for the ant-inspired selection method targeting the identification of the optimal composition solution. To identify the optimal composition solution we define a fitness function which uses the *QoS* attributes and the semantic quality as selection criteria. The proposed composition framework has been tested and evaluated on an extended version of the SAWSDL-TC benchmark.

Key words: Web service composition, semantic Web services, graph model, ant colony optimization

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction. The growing number of Web services available in public and private repositories requires fast, precise and scalable algorithms that can process them so as to present users with the most suitable functionality. Such algorithms include the automatic Web service discovery and composition as well as the selection of the optimal composition solution.

Automatic Web service discovery algorithms approach the problem of providing users with the Web services that most closely match their request, usually expressed as a list of inputs and a list of desired outputs.

Automatic Web service composition algorithms address the scenario in which the above mentioned discovery algorithms fail to return a Web service relevant to the user request because such a service is not registered in the available repositories. By Web service composition, a set of services are combined into a more complex service which satisfies the user request.

Automatic Web service discovery and composition can be achieved by combining the Web service technology with Semantic Web, thus enhancing the representation of data and processes available on the Internet with machine-interpretable information. In semantic Web service composition it is important to ensure that the results satisfy both the functional and non-functional requirements specified by the user. However, because the total number of possible composition solutions is often extremely high, an exhaustive evaluation of all the solutions is unpractical. Consequently, techniques that identify the optimal or a near-optimal composition solution without processing the entire search space are required.

In this context, our paper proposes a framework for automatic service composition which combines a composition graph model with an ant-inspired method to find the optimal composition solution. The composition graph model stores all the composition solutions that satisfy a user request. In our approach, a user request is described in terms of: functional requirements - ontological concepts that semantically describe the inputs and outputs of the requested composed service and non-functional requirements - weights associated to user preferences regarding the relevance of a composition solution's semantic quality and its *QoS* attributes. The proposed ant-inspired method, previously introduced in [3], adapts the Ant Colony Optimization (ACO) metaheuristic [1] to identify the optimal composition solution encoded in the composition graph model. We employ the following methodology for adapting the ACO metaheuristic: first, we study and analyze the biological source of inspiration of the meta-heuristic; second, we model the biological entities, relationships and processes so that they fit into our problem of selecting the optimal service composition; finally we adapt the algorithm proposed by the metaheuristic to the problem of service selection. To identify the optimal solution a fitness function which uses the *QoS* attributes and the semantic quality as selection criteria is defined. The ant-inspired composition framework has been tested and evaluated on an extended version of the SAWSDL-TC benchmark [12].

The rest of the paper is structured as follows. In Section 2 we introduce related work. The proposed framework architecture is presented in Section 3, while the associated workflows are presented in Sections 4 and 5. Section 6 discusses the adjustable parameters of the Ant-inspired composition framework and evaluates experimental results. The paper ends with our conclusions and future work proposals.

*Department of Computer Science, Technical University of Cluj-Napoca, 26-28 Baritiu str., Cluj-Napoca, Romania, ({Cristina.Pop, Viorica.Chifu, Ioan.Salomie}@cs.utcluj.ro).

2. Related Work. This section reviews some related works in the field of Web service composition.

A broker-based framework for composing Web services is described in [5]. The framework starts from a composition plan which contains a set of abstract activities. Each abstract activity is further mapped to a set of concrete services providing the same functionality but having different *QoS* attributes values. The framework identifies the optimal composition solution which provides the best global *QoS* score by using an adapted version of the HEU heuristic introduced in [11]. The main idea behind the HEU heuristic is to start from an initial feasible solution and to replace the solution elements which negatively affect the quality of the solution with other better elements. For efficiency reasons, the concept of dominance is introduced to skip the solution elements which have been previously replaced by other better elements.

In [4] an autonomous semantic Web service composition and management system inspired by the neuroendocrine-immune network is presented. The framework functionality relies on the cooperation of a set of entities to compose Web services rather than on a central coordinator. Each Web service is associated to a biological entity defined by a set of attributes, a body containing functional information and a biological behavior such as create, sleep, mutate and execute. A composed Web service satisfying a user request is represented as a network of biological entities obtained through negotiation. The negotiation between biological entities is performed using partial deduction and Linear Logic theorem proving.

An integer programming-based framework for Web service composition is proposed in [6]. The composition method used in this framework takes into consideration functional and non-functional *QoS* parameters. The framework first generates the integer-linear programming (ILP) description of the services available in a repository and then based on the user request it processes these descriptions using an ILP solver. The Web service composition solutions returned by the solver are encoded in WSBPEL.

In [8] and [9] an event calculus-based framework has been proposed as a solution for the Web service composition problem. The authors demonstrate that when a goal situation is given, the event calculus can find suitable service compositions by using the abductive planning technique. The main contributions of these works are a translation algorithm from OWL-S semantic descriptions of Web services into the event calculus and a formal framework that shows how generic composition procedures are described in the event calculus. The event calculus-based framework is presented as an alternative approach to the agent-based composition.

In the Meteor-S project [10], a Web Service Composition framework for dynamic composition of Web services has been developed which takes into consideration business constraints. The idea behind this framework is to describe the composed service as an abstract process in BPEL, and then to discover the services whose Profile matches with the defined abstract process. Once the requested service is discovered, the candidate services are selected based on the business and process constraints. The disadvantage of this approach is that the proposed technique is not totally automated.

3. Framework Architecture. This section presents the proposed Ant-inspired framework for automatic Web service composition. The aim of the framework is to: (*i*) publish semantic Web services, (*ii*) discover semantic Web services, (*iii*) compose semantic Web services, and (*iv*) select the optimal or a near-optimal composition solution according to user constraints. The conceptual architecture of the Ant-inspired framework is organized on the following layers (see Figure 3.1): the Web Service Publication layer, the Web Service Composition layer and the Composition Selection layer. Each layer supports user interaction and has an associated workflow.

Service Publication Layer. Service providers interact with the Service Publication layer through an associated graphical user interface to publish their semantic Web services in the Semantic UDDI registry. Based on the functional and non-functional information provided by the service provider, the Publication Module (*i*) generates the XML structure containing the semantic descriptions of a Web service, (*ii*) generates a *tModel* containing a reference to the XML file, (*iii*) generates the standard structures (for example *BusinessEntity* and *BusinessService*) required for storing the information about a Web service in a UDDI repository and (*iv*) groups services into clusters according to their semantic similarity. To evaluate the semantic similarity between two services, the Publication module interacts with the Service Matching module. The Publication module stores the generated structures in a Semantic UDDI registry. We chose to group services into clusters to make the discovery process more efficient.

Service Discovery Layer. Service requestors interact with the Service Discovery layer to find semantic Web services that satisfy some functional and non-functional requirements expressed using ontological concepts. In the Service Discovery layer, an important role is played by the Service Matching Module which is responsible

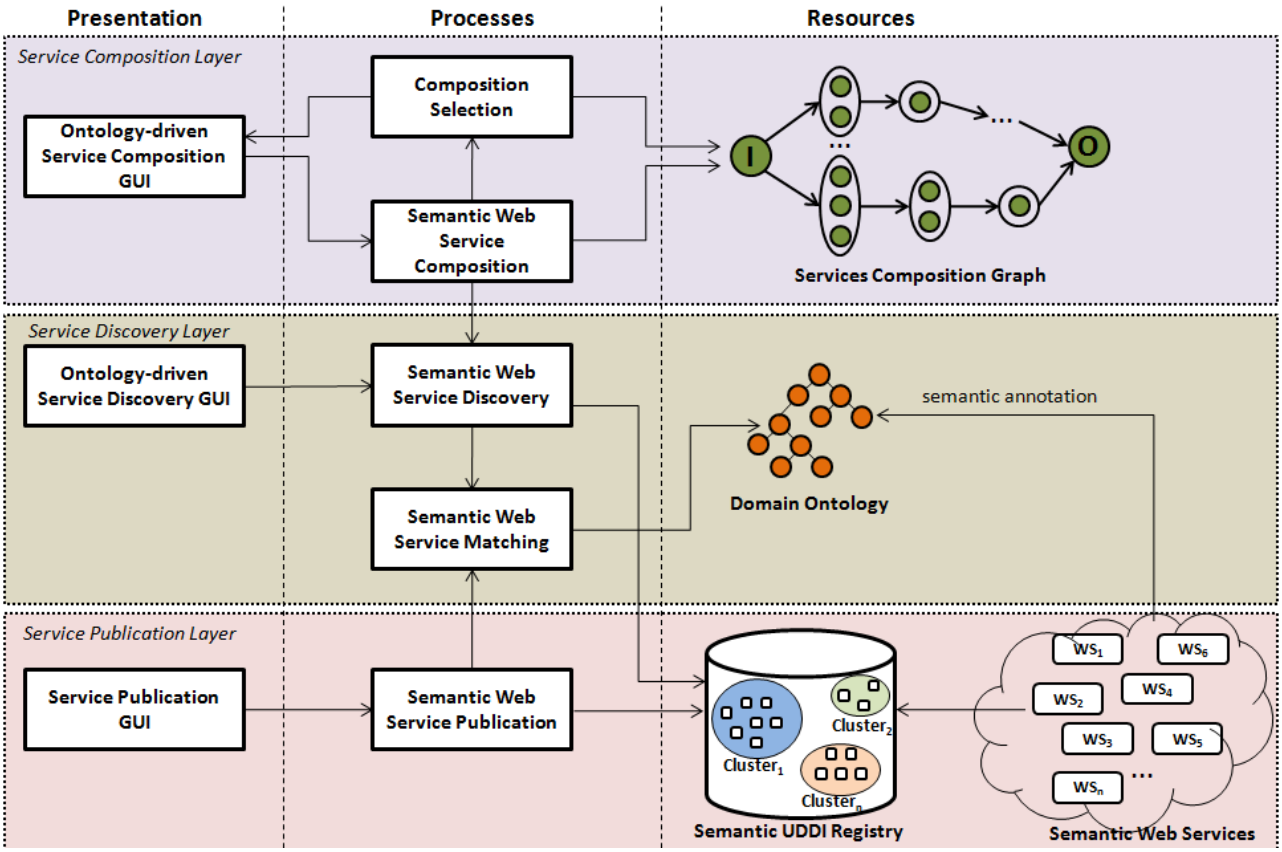

FIG. 3.1. *Framework architecture*

for evaluating the semantic similarity between two services. This module interacts directly with the ontology repository.

Service Composition Layer. The Service Composition Module interacts with the Service Discovery Module to build the composition graph which stores all the possible solutions for a specified user request. To find the optimal composition solution according to *QoS* attributes and semantic quality, the Selection Module interacts with the Service Composition Module. The Selection Module returns a ranked set of composition solutions, the first one being the optimal solution. The user is allowed to choose the solution he prefers.

4. The Web Service Composition Workflow. The main objective of the composition workflow is to obtain a graph-based representation of all the possible compositions that satisfy a user request. The composition graph will be further used as the search space in the selection process targeting the identification of the optimal composition solution according to the constraints specified in the user request.

4.1. The Composition Graph. The composition graph (see the UML representation in Figure 4.1) is automatically built for each user request.

In our approach, a *graph node* represents a cluster of similar services. A cluster contains a set of semantic similar services. We consider that two services are semantically similar if there is a degree of semantic matching (DoM), higher than a specified threshold, between them. The method for evaluating the degree of match between two services has been presented in a previous work [2]. Besides the graph nodes containing clusters of similar services we also define the input and output nodes as two special types of nodes. The input node contains a cluster with a single service which only has outputs representing a set of ontology concepts describing the user provided inputs. The output node on the other hand, contains a cluster with a single service having just inputs representing the concepts describing the user requested outputs.

A *directed edge* links a pair of cluster nodes if there is a degree of semantic match between the outputs of one of the clusters and the inputs of the other cluster.

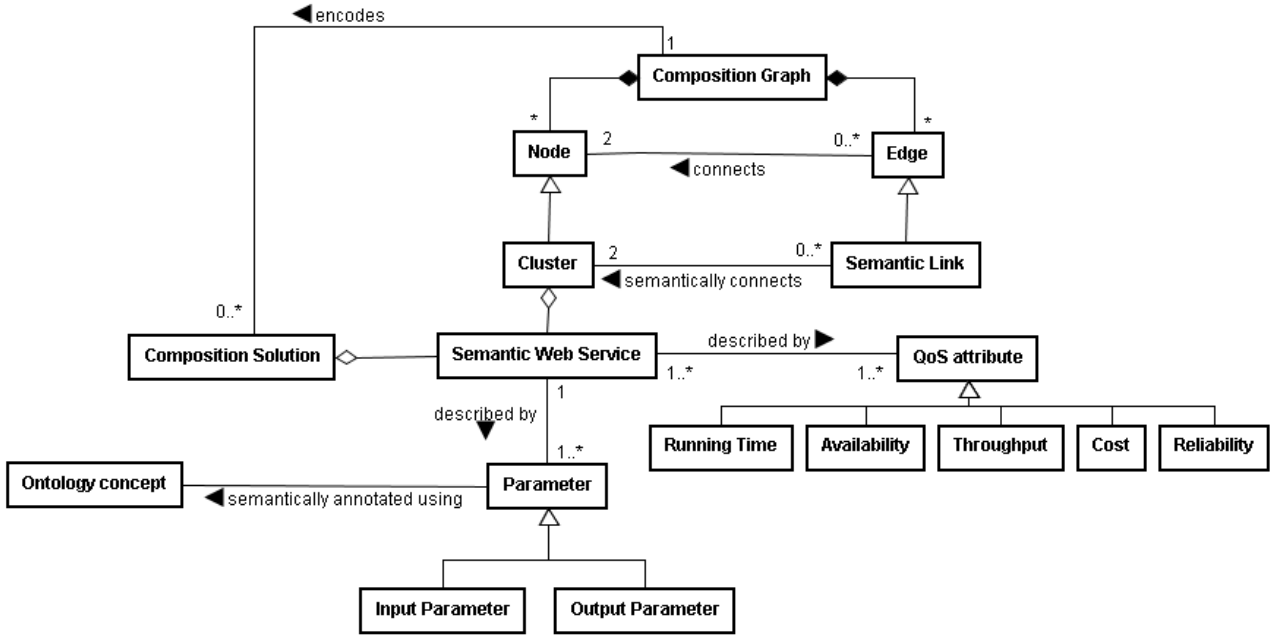

FIG. 4.1. *UML representation of the composition graph model*

The construction of the composition graph starts with a single node representing the user provided inputs. Then, the graph is expanded with new clusters of services which have a degree of match with the clusters already added in the graph. The clusters are obtained by applying a clustering method we have introduced in [2]. This way, the expansion of the graph is performed until all the inputs of the output node are satisfied. A service cluster should be added only if the clusters already present in the graph provide all the outputs required for its execution.

A *composition solution* is a directed acyclic sub-graph which contains the service of the input node, a set of services (one from each considered cluster node) and the service of the output node.

4.2. The Composition Algorithm. The composition algorithm (see Algorithm 4.2.1) takes as inputs (*i*) the user request, (*ii*) the set of services available in a repository and (*iii*) two threshold values. The threshold values are required for evaluating the semantic matching between two clusters and between two concepts.

First the composition graph is initialized (see *Initialize_Graph* in Algorithm 4.2.1) based on the user request so that it contains only the input node. Then, the algorithm iteratively performs the following two operations: (1) a set *CL* of service clusters are discovered (see *Discovery* in Algorithm 4.2.1) [2] based on the semantic matching between the concepts describing the inputs of these services and the ones describing the outputs of the services that are already in the graph, (2) for each discovered cluster *c* a corresponding node is created and added to the composition graph and as a result the set of graph edges is updated (see *Update_Edges* in Algorithm 4.2.1). A new edge is added between two nodes if there is a semantic matching between the clusters associated to these nodes.

The algorithm ends either when the user requested output parameters are provided by the services added to the composition graph or when the graph reaches a fixed point. Reaching the fixed point means that no new service clusters have been added to the composition graph for a predefined number of iterations. If the graph does not reach a fixed point means that there is at least one composition solution. In this former case, we employ a pruning process (see *Prune* in Algorithm 4.2.1), detailed in Algorithm 4.2.2, that starts from the output node and eliminates the service clusters which do not provide either (*i*) an output parameter to other services in the graph or (*ii*) an output parameter requested by the user.

The pruning algorithm (see Algorithm 4.2.2) takes a composition graph as input. The algorithm performs a breadth-first search starting from the output node in order to remove all nodes that are not on a path connecting the output node with the input node.

Regarding performance, out of the operations used in Algorithm 4.2.1, the *Discovery* procedure has the

Algorithm 4.2.1 Build_Composition_Graph

Input:
$req = (in, out)$ - the user request containing concepts that describe the provided inputs (in) and requested outputs (out);
S - set of available services;
$clTh$ - threshold for clusters matching;
cTh - threshold for concepts matching;
Output: $G = (V, E)$ - the composition graph containing a set of vertices V and a set of edges E;
begin
  v_{in} = **Generate_Input_Node**($req.in$)
  v_{out} = **Generate_Output_Node**($req.out$)
  G = **Initialize_Graph**(v_{in})
  while (($v_{out} \notin G.V$) **or** (!**Fixed_Point**(G))) **do**
    CL = **Discovery**($G.V, cTh$)
    foreach $c \in CL$ **do**
      $G.V = G.V \cup \{c\}$
      $G.E$ = **Update_Edges**($G.V, clTh$)
    end for
  end while
  if ($v_{out} \notin G.V$) **then return** $null$
  G = **Prune**(G)
  return G
end

Algorithm 4.2.2 Prune

Input: $G = (V, E)$ - the composition graph containing a set of vertices V and a set of edges E;
Output: G - the pruned composition graph;
Comments: O - the output node;
begin
  $E' = \emptyset$
  $V' = \emptyset$
  $L = \{O\}$
  while($L \neq \emptyset$)**do**
    v = **Remove_First**(L)
    $V' = V' \cup \{v\}$
    foreach $w \notin V', (wv) \in E$ **do Append**(L, w)
    foreach$(wv) \in E, w \in V', v \in V'$ **do** $E' = E' \cup (wv)$
  end while
  return (V', E')
end

highest running time: $O(N^2)$, where N is the number of clusters. In our algorithm, we call this procedure at most N times. The code in the while statements has $O(N)$ complexity in the worst case. The complexity of the pruning algorithm is $O(|V| + |E|)$. Taking all this into consideration, the complexity of Algorithm 1 is $O(N^3)$.

4.3. An Example of Building a Composition Graph. The step-by-step construction of a composition graph is illustrated in Figure 4.2. In step 1, only the input node is present; in step 2, the service clusters discovered based on the user inputs have been added; in step 3 the service clusters whose inputs are satisfied by the previously discovered clusters are added to the graph; by step 4, the entire composition graph has been built; in step 5, pruning is applied in order to remove the unnecessary parts (notice that the services S_4 and S_7 are elliminated from the composition graph). Finally, the set of possible solutions encoded in the composition graph consists of {S_1, S_2, S_5}, {S_1, S_2, S_6}, {S_1, S_3, S_5} and {S_1, S_3, S_6}.

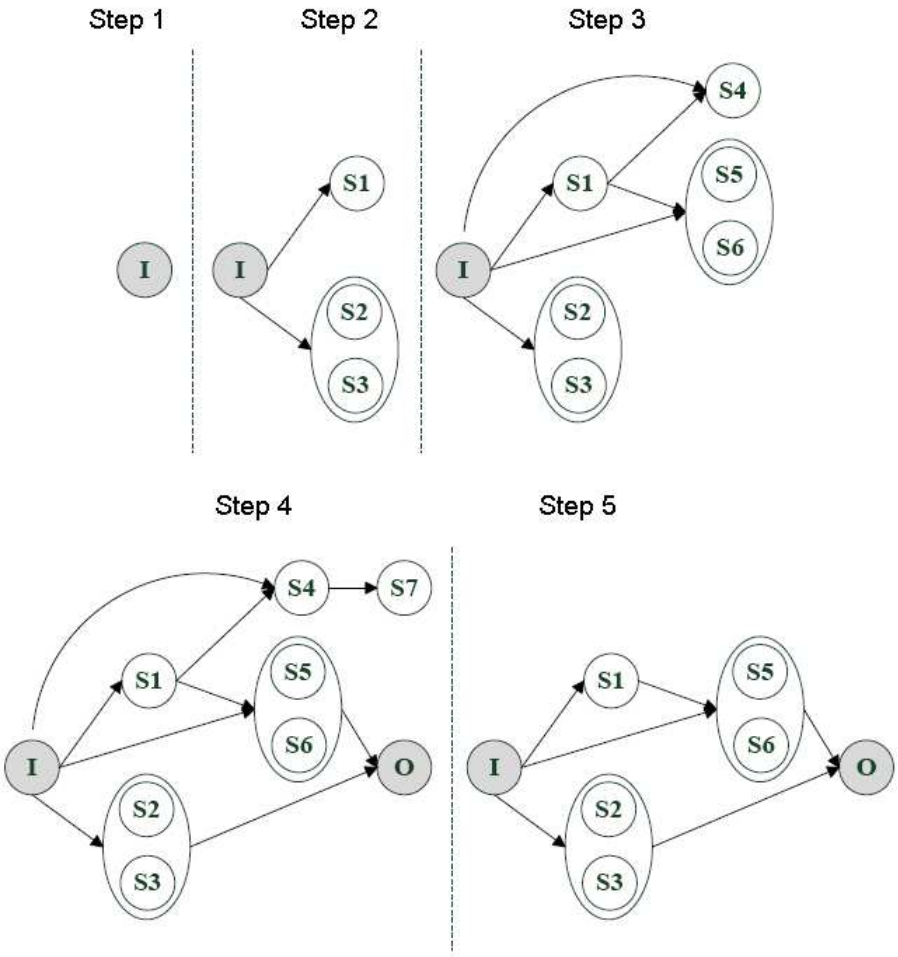

FIG. 4.2. *Composition Graph Construction*

5. The Ant-inspired Selection Workflow. For finding the optimal composition we adapted the Ant Colony Optimization (ACO) metaheuristic [1] which was proposed for solving combinatorial optimization problems. Our Ant-inspired selection workflow uses the previously described composition graph and a multi-criteria function to identify the optimal composition solution according to *QoS* user preferences and semantic quality.

5.1. Problem Formalization. The ACO metaheuristic relies on a set of artificial ants which communicate with each other to solve combinatorial optimization problems. The behavior of artificial ants is modeled according to the behavior of real ants in nature, which search for the shortest route to a food source and communicate indirectly with each other by means of the pheromone they lay on their route. In this section we present how we adapted the ACO metaheuristic to the problem of Web service composition and selection.

Just as real ants search the shortest route to a food source in a natural environment, we define a set of artificial ants that traverse the composition graph in order to find the optimal composition solution. In its search, an artificial ant can explore the composition graph and for each step it has to choose an edge that links the service where the ant is currently positioned with a new service. The choice is stochastically determined with the probability p taking into account the level of pheromone τ_{ij} (see Formula 5.2) on the edge leading to a candidate service and some other heuristic information. The heuristic information refers to the *QoS* of the candidate service at the end of the edge and the degree of semantic matching between the two services connected by the edge. The probability [1] to choose an edge (i, j) from the current service and leading to a candidate service is defined as follows:

$$p_{i,j}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}}{\sum_{c_{pq} \in N(s^p)} \tau_{pq}^{\alpha} \times \eta_{pq}^{\beta}} & \text{if } c_{pq} \in N(s^p), \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

where $N(s^p)$ is the set of candidate edges (which may be added to the partial solution), and η_{ij} is the heuristic information associated with an edge. The heuristic information is evaluated with the following formula:

$$\eta_{ij} = \frac{w_{QoS} \times QoS(S_2) + w_{Match} \times DoM(S_1, S_2)}{w_{QoS} + w_{Match}} \tag{5.2}$$

where S_1 is the current service where the ant is positioned, S_2 is the candidate service, QoS represents the QoS score of S_2, DoM is the degree of semantic matching between S_1 and S_2. The heuristic information also considers the weights w_{QoS} and w_{Match} which represent the relevance of the QoS score and of the degree of semantic matching, respectively. The QoS score of a service is computed as:

$$QoS(s) = \frac{\sum_{i=1}^{n} w_i \times Attr_i(s)}{\sum w_i} \tag{5.3}$$

where $Attr_i(s)$ represents the value of the i-th quality of service attribute, w_i represents the associated weight of the quality of service attribute and n is the number of QoS attributes considered. We use a QoS model defined in a previous work [7]. The current model considers the running time, availability, reliability, throughput and cost but it may be extended with other QoS attributes. Each of these QoS attributes needs a separate calculation method, because they have different interpretations: for some of them a higher value is better, meanwhile for others smaller values are better.

The quality of a composition solution built by an artificial ant is evaluated with the following formula:

$$Score(sol) = \frac{w_{QoS} \times QoS(sol) + w_{Match} \times Sem(sol)}{(w_{QoS} + w_{Match}) * |sol|} \tag{5.4}$$

In Formula 5.4, *QoS(sol)* represents the overall QoS of the composition solution *sol* and *Sem(sol)* represents the semantic quality score of *sol*.

QoS(sol) is computed as:

$$QoS(sol) = \frac{\sum_{i=1}^{|sol|} QoS(s_i)}{|sol|} \tag{5.5}$$

where $|sol|$ represents the number of services involved in the composition solution *sol* and *QoS(s)* is the QoS score of a service s (see formula 5.3).

The semantic quality of the composition solution *sol* is determined as:

$$Sem(sol) = \frac{\sum_{i=1}^{m} DoM(s_j, s_k)}{m} \tag{5.6}$$

where m is the number of edges contained in the composition solution *sol* and DoM evaluates the degree of semantic matching [2] between two services.

There are two types of pheromone updates, performed as suggested by the ACO metaheuristic: local update and offline update. The local update is performed by each ant at each step it makes on the composition graph according to the following formula [1]:

$$\tau_{ij} = (1 - \varphi) \times \tau_{ij} + \varphi \times \tau_0 \tag{5.7}$$

where τ_{ij} is the current pheromone level, φ is the pheromone decay coefficient, and τ_0 is the initial value of the pheromone.

The offline pheromone update is applied at the end of an ant's solution construction process as follows [1]:

$$\tau_{i,j} = (1 - \rho) \times \tau_{i,j} + \rho \times \Delta\tau_{ij} \tag{5.8}$$

where $\Delta\tau_{ij} = 1/S_{best}$, S_{best} is the best score of a composition solution found so far and ρ is the pheromone evaporation rate.

5.2. The Selection Algorithm. The selection algorithm (see Algorithm 5.2.1) takes as input a composition graph, the number of artificial ants used in the search process, the number of algorithm iterations and returns a ranked set containing the best composition solutions identified in each iteration. The algorithm adapts the Ant Colony Optimization (ACO) metaheuristic [1]. Like most ACO algorithms, the selection algorithm iteratively searches for the best composition solutions.

Algorithm 5.2.1 Ant-based_Selection

Input:
$noIt$ - the number of iterations;
$noAnts$ - the number of artificial ants;
$G = (V, E)$ - the composition graph;
Output:
SOL - the set of the best composition solutions;
begin
  $SOL = \emptyset$
  $noItModif = 0$
  $Ants =$ **Initialize_Ants**$(noAnts)$
  while $(noItModif < noIt)$ **do**
    $noItModif = noItModif + 1$
    $max = 0$
    foreach $a \in Ants$ **do**
      $result =$ **Find_Solution**(G, a)
      $minScore =$ **Find_Min_Score**(SOL)
      $maxScore =$ **Find_Max_Score**(SOL)
      if $(minScore <$ **Score**$(result))$ **then**
        $SOL =$ **Remove_Worse**(SOL)
        $SOL =$ **Add**$(SOL, result)$
        $noItModif = 0$
      end if
      if $(maxScore <$ **Score**$(result))$ **then**
        $max =$ **Score**$(result)$
        $maxSol = result$
      end if
    end for
    $G =$ **Apply_Offline_Pheromone**$(G, maxSol)$
  end while
  return SOL
end

Within an iteration, a set of artificial ants traverse the composition graph searching for composition solutions. In the first step, an ant is randomly placed on a graph node. Then, the ant will try to find input concepts for the node it is currently on. To do this, it will go backwards on an edge. When positioned on another node, the ant will try again to find appropriate inputs for that node. This operation will be performed until there are no more services whose inputs have no correspondents to outputs of other services. Once this is done, the ant will pick an edge in a forward direction from its current position node trying to move towards the output

node. From then on, the ant will pick edges outgoing from nodes that are not providing inputs to any node which is already present in the partial solution. Once a new node is picked, the ant will again go on back edges searching to provide all its inputs. This process is repeated until the ant reaches the output node and all the inputs of all the nodes are satisfied. The algorithm avoids adding a new edge if a cycle is about to be created in the composition graph. After all the ants manage to obtain a composition solution, the offline pheromone update is applied (see *Apply_Offline_Pheromone_Update* in Algorithm 5.2.1) on the edges of the best solution found so far. The best solution is identified based on Formula 5.4 (see *Score* in Algorithm 5.2.1). Finally, the algorithm returns the list of the best composition solutions found in each iteration.

Algorithm 5.2.2 illustrates how an artificial ant builds a solution. First, an empty solution is initialized. Then a random node is chosen followed by choosing an appropriate service from the corresponding cluster. The selected service is added to the partial solution. Then, all the services providing the required inputs for the partial solution are backward searched (see *Find_Inputs* in Algorithm 5.2.2). If the output node is not yet present in the partial solution, the algorithm performs a forward search to find another service (see *Find_Next_Service* in Algorithm 5.2.2) for which the current partial solution can provide inputs. The algorithm finally returns the found solution.

Algorithm 5.2.2 Find_Solution

Input: G - the composition graph
Output: *sol* - the found solution
Comments:
$Choose\_Random\_Node$ - picks a random graph node
$Pick\_Service$ - chooses a service from a cluster
$Is\_Solution$ - evaluates the partial solution
begin
  $sol = \emptyset$
  $N =$ **Choose_Random_Node**(G)
  $N.chosenService =$ **Pick_Service**(N)
  $sol =$ **Add_Node**(sol, N)
  while (**Is_Solution**$(sol) == false$) **do**
    $sol =$ **Find_Inputs**(sol, N)
      if (**Contains**$(sol, O) == false$) **then**
        $N =$ **Find_Next_Service**(sol)
      end if
  end while
  return *sol*
end

Algorithm 5.2.3 finds the services that provide inputs for the services that are contained in a partial solution. The algorithm is used recursively until all the inputs are provided. For a service in the partial solution, the algorithm first iterates through all the inputs which are not yet provided. It then finds a providing service, such that by adding the corresponding edge the graph remains acyclic. This service is chosen stochastically (see Formula 5.1), with the probabilities given by the pheromone level and heuristic information of the edges connecting the current service and the considered one. Then, the algorithm adds the service and the edge to the partial solution if they are not already present. These steps are repeated recursively until a partial composition solution is found.

The *Find_Next_Service* procedure in Algorithm 5.2.2 which is detailed in Algorithm 5.2.4 picks a new service to add to a partial solution with all the inputs provided. First, it determines the services which should provide inputs for the newly added one (see *Get_Nodes* in Algorithm 5.2.4). Since we want to go towards the output node, we will pick those services in the partial solution which are not already providing inputs for other services in it. We then determine the set of edges to consider: all the outgoing edges with the start service in the set previously determined, and the end service not in our partial solution (see *Get_Edges* in Algorithm 5.2.4). We then call a procedure that will choose an edge stochastically based on the pheromone levels and the heuristic information (see *Pick_Edge_Stochastically* in Algorithm 5.2.4). Next we add the edge and the end service to our solution and apply the local update procedure (see *Local_Update* in Algorithm 5.2.4).

Algorithm 5.2.3 Find_Inputs

Input:
sol_{part} - a partial composition solution
s - the service where the ant is currently positioned
Output: sol_{part} - the updated partial composition solution
begin
  foreach $a \in Ants$ **do**
    repeat
      $s' =$ **Find_Providing_Service**$(s.in)$
    until(**Edge**$(s, s') \notin Edges(sol_{part})$**or**
      Acyclic$(sol_{part} \cup$ **Edge**$(s, s')) == true)$
    if $(Edge(s, s') \notin Edges(sol_{part}))$ **then**
      $sol_{part} =$ **Add_Service**(sol_{part}, s')
      $a =$ **Pick_Service**(s')
    end if
    Local_Update(**Edge**(s, s'))
  end for
  if (**Unprovided_Inputs**$(s') \neq \emptyset$) **then**
    $sol_{part} =$ **Find_Inputs**(sol_{part}, s')
  end if
  return sol_{part}
end

Algorithm 5.2.4 Find_Next_Service

Input: sol_{part} - the partial solution; G - the composition graph;
Output: sol_{part} - the updated partial solution;
begin
  $candidateNodes =$ **Get_Nodes**(G, sol_{part})
  $candidateEdges =$ **Get_Edges**$(G, CandidateNodes, sol_{part})$
  $edge =$ **Pick_Edge_Stochastically**$(candidateEdges)$
  $sol_{part} = sol_{part} \cup \{$**Get_End_Node**$(edge), edge\}$
  $edge =$ **Local_Update**$(edge)$
  return $edge$
end

In the worst case scenario, when constructing a solution the entire composition graph is traversed, so the complexity of Algorithm 5.2.2 is $O(V + E)$. Based on this, the complexity of Algorihtm 5.2.1 may be estimated. Its complexity is $O(N \times A \times (V + E))$, where N is the number of iterations and A is the number of ants. A will generally be a small constant. However, in the worst case scenario, N can be as high as the number of possible solutions. In practice, as it will be shown in Section 6, this number is much lower, and $N \times A$ is often about 5% of the number of possible solutions. An alternative to this implementation (should the worst-case scenario be unacceptable) is to run for a fixed number of iterations. The algorithm is expected to perform well in such a case too, but care should be taken when fixing the number of iterations, as too few may result in bad solutions and too many will result in extra work. While in theory this will lead to a better complexity, in practice the currently chosen method works better.

5.3. An Example of Selecting a Composition Solution. Figure 5.1 presents the way in which an ant selects a composition solution. For simplicity, we suppose that there is only one service in each cluster and that at the beginning of the iteration the pheromone levels are the same on all the edges. The composition graph can be seen in subfigure 5.1-(a). In the first step, the initial random node is chosen - suppose it is S_3 (subfigure 5.1 - (b)). S_3 needs providers for its inputs, so it calls one of its providers in the composition graph. Such a node is S_2, which can actually provide all of the required inputs (see subfigure 5.1 - (c)).

The ant selects it and searches for a provider for the newly added node. The node I is the only possibility because it provides all the inputs for S_2, so it is added to the solution (subfigure 5.1 - (d)). The ant can now go

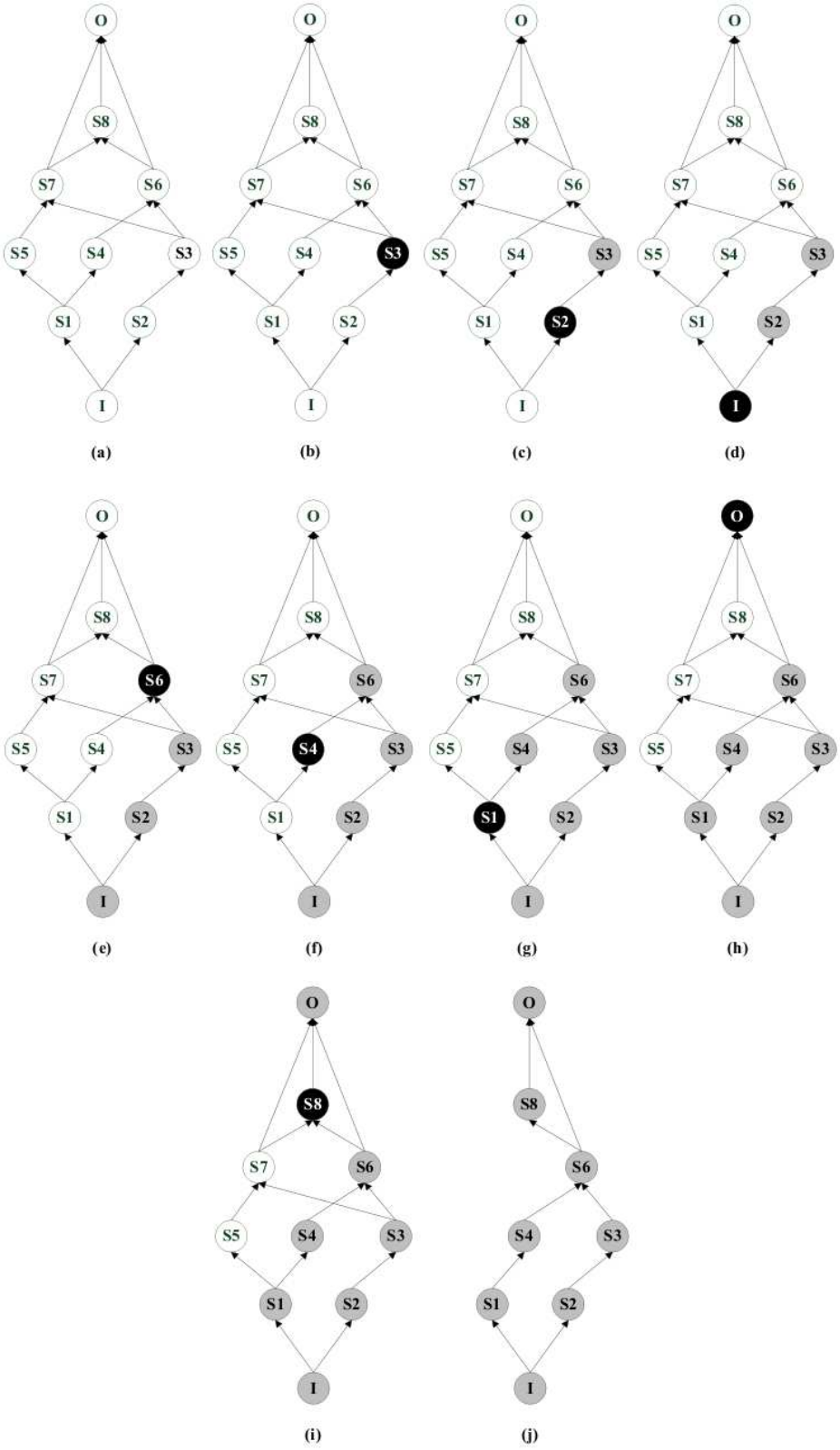

FIG. 5.1. *Example of Selecting a Composition Solution*

further, by looking for a node which requires the inputs that the current partial solution can provide. One such node is S_6 (subfigure 5.1 - (e)). Another possible choice would have been S_7, but the ant notices that S_6 has a better semantic matching. Also, the ant prefers S_6 to S_5 because its providers are not already supplying for nodes currently in the partial solution. The ant notices that S_4 can provide all the required inputs and therefore the ant adds the service to the partial solution (subfigure 5.1 - (f)). The ant looks again for nodes which can provide inputs for S_4, and it picks S_1 (subfigure 5.1 - (g)). The ant searches for inputs for S_1 and finds I, which is already in the partial solution. Then the ant goes forward again, and it can choose from S_8 and O. Suppose it picks O (subfigure 5.1 - (h)) - because only part of its inputs are provided, the ant needs to search for nodes which can provide the others. S_8 is such a node, so the ant picks it (sub-figure 5.1 - (i)). Now the ant needs to search for inputs for S_8, and S_6 already provides them. S_6 can provide all the inputs necessary and is already in the solution, so no providers for its inputs need to be searched. Therefore, since both I and O are in the sub-graph and there are no free inputs, the ant can consider this to be a solution and stops searching (subfigure 5.1 - (j)). The score of the identified solution is computed, and, if necessary, the pheromones are updated.

6. Evaluation and Testing. This section describes the evaluation and testing methodology applied on the Ant-inspired composition framework as well as the way some adjustable parameters have been tuned.

6.1. Evaluation Methods. The main criteria according to which the Ant-inspired composition framework has been evaluated are its correctness and its efficiency. These framework properties will be expressed numerically in order to have an accurate estimation. To evaluate how well the framework works, we have used the following criteria: the quality of the provided composition solutions and the framework efficiency.

The quality of the provided composition solution is measured by counting how many solutions in the list of k best found solutions should actually be there, i.e. how many of the best k possible composition solutions were found by the selection algorithm. The ratio between the number of such solutions found and k will show how correct the selection algorithm is.

To evaluate the efficiency of the composition framework, we counted the number of composition solutions encoded in the composition graph and compared this number to the total number of solutions returned by the selection algorithm for a given user request. The number of iterations has been set so that at least 50% of the best possibile k solutions were found by the selection algorithm. The smaller the ratio between the selected number of solutions and the total number of solutions, the more performant our approach is.

6.2. Test Service Collection. We have tested our ant-based service composition technique on SAWSDL-TC [12], a benchmark containing 894 services, proposed for evaluating the performances of service matchmaking algorithms. This benchmark contains 894 Web services which are semantically annotated using concepts from several domain ontologies according to the SAWSDL specification. The services belong to the following domains: education, medical care, food, travel, communication, economy and weapon. In order to obtain a more complex composition scenario, a few extra services were added to the original set. Also, some services in the initial set were removed, as they referenced ontologies that were no longer available on the Web. In total, the set we worked with had around 700 services in it.

Because the semantic Web services from the SAWSDL-TC benchmark lack *QoS* attributes, we associated to each service a *QoS* description. When choosing the range (see Table 6.1) of the *QoS* values we were inspired by [13]. In computing the final score of a composition solution, each *QoS* parameter was normalized and received a score between 0 and 1, so that none of them would be given more importance than others by default.

TABLE 6.1
QoS Value Ranges

QoS Parameter	**Range**	**Evaluation**
Response Time	50 - 2000	(2000 - value)/2000
Availability	50 - 98	value/100
Throughput	1 - 12	value/12
Reliability	60 - 75	value/100
Cost	0 - 100	value/100

6.3. Test Scenario. To evaluate our Ant-inspired service composition framework we have chosen a scenario from the medical domain. This scenario refers to a typical request from a person (identified by name,

address, country), which has certain symptoms and wants to go to a particular doctor. As a result, the person will be assigned to a hospital room (indicated by city, hospital name and room number) at a certain date. For the considered scenario the user request is presented in Table 6.2.

TABLE 6.2
User Provided Parameters

Inputs	Outputs
PatientOntology.owl#PersonName	PatientOntology.owl#City
PatientOntology.owl#Address	PatientOntology.owl#Hospital
PatientOntology.owl#Country	PatientOntology.owl#Room
PatientOntology.owl#Symptom	PatientOntology.owl#TransportNumber
PatientOntology.owl#Physician	PatientOntology.owl#DateTime

6.4. Parameter Tuning. By analyzing the experimental results we noticed that there are two adjustable parameters which strongly influence the composition method, namely the *cluster threshold* and the *concept threshold*. Based on our experimental results, the most appropriate value for the cluster threshold *clTh* proved to be 0.4. If this threshold value is lower, there will be too many edges, and if it is too high, there will be too few edges in the graph. The concept threshold *cTh* is used to determine if two concepts are similar enough to consider them as representing the same thing. This threshold is used both in the discovery method and in the edge generation procedure. Taking into account the values returned by the matching module, 0.25 was chosen as an appropriate value for the concept threshold parameter.

In the case of the selection method, the most important are the parameters related to pheromone evolution. A first parameter, the *initial evaporation*, is used to initialize the pheromone levels on the edges of the graph. It is currently set to 1, but as long as its value is strictly positive it does not influence the outcome in a significant way. Another parameter is the *pheromone evaporation rate*. This is the parameter used in the offline update, and determines the evolution of the pheromone from one iteration to the next one. We found that a value which allows the pheromones to vary in a satisfactory manner is 0.15. A third parameter is the *pheromone decay coefficient* used in the local pheromone update, which is applied by each ant as its constructs a solution on each edge it traverses. Therefore, its values must be smaller than that of the previous parameter, otherwise it would have a too large weight in pheromone evolution. We found that 0.05 is an appropriate value. Besides the parameters related to pheromone evolution, the number of ants and the number of iterations in which the global optimal solution has not changed should be tuned. Based on our experimental results we have set the number of ants to 4 and the number of iterations to 3.

6.5. Results Analysis. An example of services involved in the composition and selection processes for the considered scenario (see Table 6.2) is provided in Tables 6.3 and 6.4. All the concepts describing these services are part of a *patient ontology* from the SAWSDL-TC benchmark.

TABLE 6.3
Example of Web service 1

Diagnosis_TaxedPriceCostAndHealingPlan_Service.wsdl		
Service ID: 3	**Inputs**	**Outputs**
Cluster ID: 2	Diagnosis	TaxedPrice CostAndHealingPlan
QoS Parameters		
Running Time: 1367	Availability: 74	Throughput: 6
Cost: 93	Reliability: 69	

In our experiments, we have set all the weights (see Formula 5.4) used to calculate the score of a solution to 1. This way, we give the same importance to the *QoS* score and the semantic matching score. Also, within the *QoS* score computation, all the attributes count the same. For these parameters, the calculated optimal composition solution has a score of 0.8459. The algorithm returned four (see Table 6.5) of the possible five best results. That means that we had a success rate of 0.8. Also, out of the 1200 possible compositions, only about

TABLE 6.4
Example of Web service 2

Patient_HealthInsuranceNumberCommercialOrganisation_Service.wsdl		
Service ID: 13	**Inputs**	**Outputs**
Cluster ID: 3	Patient	HealthInsuranceNumber
		Commercial Organization
QoS Parameters		
Running Time: 1169	Availability: 65	Throughput: 6
Cost: 73	Reliability: 68	

90 were generated. In order to have a success rate of 0.5, 60 generated solutions would have been enough. That means that only a fraction 0.05 of the total number of solutions need to be generated to have relevant results. For about 100 iterations, we had the best solution among the list of retrieved solutions in more than 80% of the time. This leads us to the conclusion that the method achieves its purposes: provides the user with good composition solutions with a small computational cost.

7. Conclusion. In this paper, we propose an Ant-inspired Web service composition framework which uses a composition graph model combined with an Ant-inspired method to find the optimal service composition solution. The composition graph model stores all the composition solutions that satisfy a user request. The Ant-inspired method identifies the optimal or a near-optimal composition solution encoded in the composition graph model. The composition solutions are evaluated with a fitness function which considers the *QoS* attributes and the semantic quality as selection criteria. The ant-inspired composition framework has been tested and evaluated on an extended version of the SAWSDL-TC benchmark [12].

As future work, we intend to speed-up the selection algorithm by parallelizing the ants' searching process.

REFERENCES

[1] M. DORIGO, M. BIRATTARI, AND T. STTZLE, *Ant Colony Optimization, Artificial Ants as a Computational Intelligence Technique*, The IEEE Computational Intelligence Magazine, (2006).
[2] C. B. POP, V. R. CHIFU, I. SALOMIE, M. DINSOREANU, T. DAVID, AND V. ACRETOAIE, *An Ant-inspired Approach for Semantic Web Service Clustering*, Proceedings of the 9th Roedunet IEEE International Conference, (2010), pp. 145-150.
[3] C. B. Pop, V. R. Chifu, I. Salomie, M. Dinsoreanu, T. David, and V. Acretoaie, *Ant-Inspired Technique for Automatic Web Service Composition and Selection*, Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, (2010), pp. 449-455.
[4] Y. DING, H. SUN, AND K. HAO, *A bio-inspired emergent system for intelligent Web service composition and management*, Knowledge-Based Systems Journal, Volume 20, Issue 5, (2007)
[5] L. YUAN-SHENG, ET AL., *An Improved Heuristic for QoS-aware Service Composition Framework*, Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications, (2008), pp. 360 - 367.
[6] J. JUNG-WOON YOO, S. KUMARA, AND D. LEE. *A Web Service Composition Framework Using Integer Programming with Non-Functional Objectives and Constraints*, Proceedings of the 10th Conference on E-Commerce Technology and the Fifth Conference on Enterprise Computing, E-Commerce and E-Services, (2008), pp. 347 - 350,
[7] V. R. CHIFU, C. B. POP, I. SALOMIE, M. DINSOREANU, A. E. KOVER, AND R. VACHTER, *Web Service Composition Technique Based on a Service Graph and Particle Swarm Optimization*, Proceedings of the 2010 IEEE International Conference on Intelligent Computer, Communication and Processing, Cluj-Napoca (Romania), (2010), pp. 265-272.
[8] O. AYDIN, N.K. CICEKLI, AND I. CICEKLI, *Towards Automated Web Service Composition with the Abductive Event Calculus*, Applications of Logic Programming in the Semantic Web and Semantic Web Services, Seattle, (2006).
[9] O. AYDIN, N. K. CICEKLI, I. CICEKLI, *Automated Web Services Composition with the Event Calculus*, International Workshop in Engineering Societies in the Agents World, Athens, (2007).
[10] Z. WU, ET. AL., *Automatic Composition of Semantic Web Services using Process and Data Mediation - Technical Report*, (2007).
[11] M.M.AKBAR, E.G.MANNING, G.C. SHOJA AND S.KHAN, *Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem*, International Conference on Computational Science, (2001) LNCS 2074.
[12] SAWSDL-TC, http://projects.semwebcentral.org/projects/sawsdl-tc/
[13] The QWS Dataset, http://www.uoguelph.ca/ qmahmoud/qws/index.html

Edited by: Dana Petcu and Alex Galis
Received: March 1, 2011
Accepted: March 31, 2011

TABLE 6.5
Top 4 composition solutions for the considered scenario

Solution number	**The services in the solution**	**Solution score**
1	PersonNameAddressCountry_Patient_Service; SymptomPhysician_Diagnosis_Service; Patient_HealthInsuranceNumberOrganisation_Service; PatientDiagnosis_HealthInsuranceNumberInsuranceCompany TaxFreePriceCostAndHealingPlan_Service; Diagnosis_TaxedPriceCostAndHealingPlan_Service; TaxedPriceCostAndHealingPlanHealthInsuranceNumber InsuranceCompany_CityHospitalRoomDateTime; CityHospitalRoomDateTime_TransportNumber_Service	0.8459
2	PersonNameAddressCountry_Patient_Service; SymptomPhysician_Diagnosis_Service; PatientDiagnosis_HealthInsuranceNumber CommercialOrganization TaxFreePriceCostAndHealingPlan_Service; Patient_HealthInsuranceNumberInsuranceCompany_Service; Diagnosis_TaxedPriceCostAndHealingPlan_Service; TaxedPriceCostAndHealingPlanHealthInsuranceNumber InsuranceCompany_CityHospitalRoomDateTime; CityHospitalRoomDateTime_TransportNumber_Service	0.8433
3	PersonNameAddressCountry_Patient_Service; SymptomPhysician_Diagnosis_Service; Diagnosis_TaxedPriceCostAndHealingPlan_Service; PatientDiagnosis_HealthInsuranceNumber CommercialOrganization TaxFreePriceCostAndHealingPlan_Service; TaxedPriceCostAndHealingPlanHealthInsuranceNumber InsuranceCompany_CityHospitalRoomDateTime; CityHospitalRoomDateTime_TransportNumber_Service	0.8422
4	PersonNameAddressCountry_Patient_Service; SymptomPhysician_Diagnosis_Service; Patient_HealthInsuranceNumberOrganisation_Service; Diagnosis_TaxedPriceCostAndHealingPlan_Service; PatientDiagnosis_HealthInsuranceNumber CommercialOrganization TaxFreePriceCostAndHealingPlan_Service; TaxedPriceCostAndHealingPlanHealthInsuranceNumber InsuranceCompany_CityHospitalRoomDateTime; CityHospitalRoomDateTime_TransportNumber_Service	0.8422