Ant-inspired Technique for Automatic Web Service Composition and Selection

Cristina Bianca Pop, Viorica Rozina Chifu, Ioan Salomie, Mihaela Dinsoreanu, Tudor David, Vlad Acretoaie
Department of Computer Science
Technical University of Cluj-Napoca
26-28 Baritiu str., Cluj-Napoca, Romania
Email: {Cristina.Pop, Viorica.Chifu, Ioan.Salomie, Mihaela.Dinsoreanu}@cs.utcluj.ro

Abstract—This paper presents a new technique for semantic Web service composition inspired by the behavior of ants. The proposed technique combines a service composition graph model with the ant colony optimization metaheuristic to identify the optimal composition solution. The following criteria are considered to select the optimal composition solution: the QoS attributes of the services and the semantic quality of the connections between the services involved in a composition solution.

I. INTRODUCTION

Web services have become a popular method for exposing software module capabilities to a large number of potential users. They are platform-independent software components, and are available in a distributed environment, generally on the Internet. Web services have a standardized description in the form of a WSDL file, which allows them to be discovered and invoked. This also allows other applications to directly interact with Web services without any human intervention. One of the features that have made Web services popular is the fact that they can be reused, both to achieve their standalone functionality and to form larger and more complex web services by composing them. Such composite Web services can be obtained either through direct human intervention, or through an automatic approach, the latter being preferred.

To automatically discover and compose Web services with good results, one often requires more than the plain syntactic information offered by the WSDL file. A solution is to employ semantic descriptions of Web services. This means assigning concepts from ontologies to the inputs, outputs and even the functionalities of services, thus giving them more meaning. Once the services are composed they must be invoked and the results they provide must be passed from one to another to allow the invocation of all the services in a composition. One of the most important issues in automatic Web service composition is ensuring that the results are as good as possible from the point of view both of the semantic matching between the services and the values of the quality of service parameters in the solution. However, the total number of possible solutions is often extremely high, and an exhaustive evaluation of all the solutions would be unpractical, as users often expect the results in a matter of seconds. Also, such a system needs to be able to work under heavy loads and with repositories holding a very large number of services. It is therefore clear that this is an area where a trade-off between the running time and the result quality exists.

In this context, this paper proposes an efficient method for automatically composing semantic Web services as well as a method for selecting the optimal composition solution. The main contributions of these methods are a composition graph that encapsulates the entire set of composition solutions and the adaptation of the Ant Colony Optimization Metaheuristic to the problem of selecting the optimal composition solution.

The paper is structured as follows. Section II introduces related work. Section III presents the method for automatic Web service composition, while section IV describes the method for selecting the optimal composition solution. Section V highlights experimental results. We end our paper with conclusions and future work proposals.

II. RELATED WORK

The existing approaches for Web service composition are based either on static or dynamic composition methods. According to [1], the approaches currently being investigated in the area of static composition are Web-service orchestration, which composes services by employing a central coordinator which is responsible for composing and invoking individual simple services. The second approach based on dynamic composition methods is called Web services choreography and defines complex tasks and the conversations to be undertaken by the participants. In this section we mostly focus on dynamic approaches, most of them surveyed in [3].

Some dynamic approaches are based on BPEL4WS, which is an XML-based language used for describing Web service choreographies according to specified business rules. The downsides of BPEL are that it does not address conformance and QoS and that it deals with connectivity only, not with correctness. Other methods are based on π-calculus which is an algebraic method of describing processes. In this approach, the basic entity is the process, which has several sub-types such as empty process, I/O processes, parallel composition and recursive definition. This formalism allows to reason about composition correctness. Another dynamic service composition approach is based on Petri nets. Petri nets are directed, connected and bipartite graphs in which nodes represent places and transitions and tokens occupy places. When in every place connected to a transition there is a token, that transition is enabled (for a Web service, this corresponds to
all the inputs being provided). An enabled transition fires by removing one token from every input place, and deploying tokens to the output places. The next method we mention is model checking, which is used to formally verify finite-state concurrent systems. In case of model checking, the system specifications are described using temporal logic, and then the model is used (traversed forward and back) to determine whether the specification holds. Finally, we have the semantic-based service composition methods which are based on the semantic information added to Web services, information which is described in ontologies. The main advantage of semantic-based composition is that it eases automatic Web service composition, meaning that no human involvement is required. This method does not allow correctness verification, like some others do, but the quality of the results may be ensured through a careful design on the algorithms. It also generally ensures a good scalability of the service composition operation.

Current approaches on semantic-based composition aim to apply biologically-inspired methods. An attempt has been made in [5] to use ant colony algorithms in the context of service composition. The authors consider a composition to be a fixed linear sequence of services. For each of the services forming the composition, several candidates are considered, each of the candidates for a service slot having the same function. Such a composition is evaluated based on the QoS of the services. The authors use optimal chaos and ant colony optimization to determine the best choice for each service. This approach does not take into account semantic information, and is not able to deal with a more complex structure of the composition.

### III. THE COMPOSITION METHOD

The main objective of the composition method is to obtain a graph-based representation of all the possible compositions that would lead to the required result. This graph will be further used as the search space for a selection method targeting the identification of the optimal composition solution according to the constraints specified in the user request.

#### A. The Composition Graph

The composition graph is composed of nodes and edges that link these nodes. A graph node represents a cluster of similar services. A cluster is an entity which contains a set of services, with the property that there is a high degree of semantic matching between each pair of services in the cluster. One aspect that needs to be considered is how restrictive the definition of a cluster needs to be. We should allow services with similar functionalities, inputs and outputs to be part of the same cluster. Also, we might consider placing services with the same functionality but a different number of inputs and outputs in the same cluster.

Besides the nodes containing a cluster of similar services there are two other special types of nodes in the composition graph that need to be mentioned: the input and output nodes. The input node represents 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, represents a cluster with a single service having just inputs representing the concepts describing the user requested outputs.

A directed edge can be used to link a pair of clusters if there is a high degree of match between the outputs of one of the clusters and the inputs of the second cluster. We presented the method for evaluating the degree of match in a previous work [4].

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 that are good semantic matches for the clusters already in the graph. These clusters are obtained by applying a clustering method we introduced in [4]. 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 sufficient outputs for its execution. When adding a node, all the possible edges going from the nodes already in the graph towards the newly added one are considered.

A composition solution derived from the composition graph is a directed acyclic graph having services as nodes, such that for each input of each service there is another service in the solution providing that input via a directed edge. The directed graph must be acyclic to avoid situations when two nodes each requiring concepts from one another are added as such in the solution. The solution must contain the output node, and there must be a path from each node in a solution to the output node. The last constraint is imposed to avoid solutions that have unnecessary nodes.

#### B. The Composition Algorithm

The composition algorithm Algorithm 1 takes as input the thresholds required for evaluating the matching between clusters, concepts as well as missing concepts and outputs the composition graph. Lines 1 and 2 initialize the graph, so that it contains only the input node and no edges. In lines 3 through 7 we iteratively (i) call the discovery procedure [4] to find clusters whose inputs are all provided by the existing nodes, (ii) add them to the graph, and then (iii) add edges from all the pre-existing nodes which can provide inputs to the newly added node. We label the edges with the degree of match between the inputs of the added node and the outputs of the existing ones. Next, in lines 8 to 12, we repeat this procedure, this time also allowing a small variation in the number of inputs/outputs, as discussed in the previous sub-section. In line 13, we test if the output node is present in the composition graph. If it is not present, it means that there is no solution to the composition problem. Finally, in line 14, we prune the unnecessary parts of the graph. In the last line, we return the obtained composition graph.

There are several important procedures called in Algorithm 1. First, on line 3 and 8 we have the Discovery procedure. This procedure takes the set of nodes in the graph and a concept
Algorithm 1: BuildGraph

Input: Cluster_thres - threshold for clusters matching;
   Concept_thres - threshold for concepts matching;
   Missing_thres - threshold for missing concepts

Output: G - the composition graph
1  V = \{I\};
2  E = \{\}\;
3  while L' = Discovery(V,Concept_thres,0) \neq \{\} do
4      foreach cluster C \in L' do
5          V = V \cup \{C\};
6          E = E \cup \{(C' \mid DoM(C',C) > Cluster_thres, C' \in V) \cup \{(C' C) DoM(C,C') > Cluster_thres, C' \in V\};
7          label(CC') = DoM(C,C');
8  while L' = Discovery(V,Concept_thres,Missing_thres) \neq \{\} do
9      foreach cluster C \in L' do
10         E = E \cup \{(C' C) DoM(C',C) > Cluster_thres, C' \in V) \cup \{(C' C) DoM(C,C') > Cluster_thres, C' \in V\};
11         label(CC') = DoM(C,C');
12  if O \notin V then return false;
13  (V,E) = Prune(V,E,O);
14  return G = (V,E);

threshold as inputs, and returns a set of nodes that have not been previously added to the graph. These new nodes have the property that for each of their inputs there exists an output available in the graph, such that the degree of match between the two concepts is at least as big as the threshold given as a parameter. There is a third parameter to this procedure, which allows to add clusters that have only a small portion of non-invokeable services. Another procedure is the DoM procedure, which computes how similar two clusters are among them. Finally, we have the Prune procedure, which is shown in Algorithm 2. The Prune algorithm takes a graph given by its edges and vertices as inputs, and a node from which the search starts. That node must be the output node. The algorithm starts from that node, and goes on backward edges only, to reach all the nodes in the graph we can. This is basically a breadth-first search. These nodes are added to a newly created set of nodes. At the end, the algorithm adds to the new set of edges only those initial edges which connected two nodes reached by the algorithm. By calling this procedure, all the nodes not having a path to the output node are eliminated.

Regarding performance, out of the operations used in Algorithm 1, the Discovery procedure has the 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

Algorithm 2: Prune

Input: (V,E) - the nodes and edges of the constructed graph; O - the output node

Output: (V',E') - the pruned graph
1  E' = \{\}\;
2  V' = \{\}\;
3  L = \{O\};
4  while L \neq \{\} do
5      v = removeFirst(L);
6      V' = V' \cup \{v\};
7      foreach w \notin V', (wv) \in E do
8          append(L,w);
9      foreach (wv) \in E, w \in V' v \in V' do
10         E' = E' \cup (wv);
11  return (V',E');

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 \(O(N^3)\).

IV. THE ANT-INSPIRED SELECTION METHOD

In this subsection we present a method for selecting the optimal composition solution. The selection method adapts the Ant Colony Optimization metaheuristic \cite{2} to the problem of Web service composition.

A. Problem Formalization

In what follows we describe the mapping of the Ant Colony Optimization metaheuristic to the optimal service composition selection problem. In \cite{2}, authors present the model of a combinatorial optimization problem on which the Ant Colony metaheuristic can be applied. This model is represented as a triple \(P = (S; \Omega; f)\), where (i) \(S\) is the search space, (ii) \(\Omega\) is a set of constraints applied on the elements of the search space and (iii) \(f : S \rightarrow R^+\) is an objective function that needs to be minimized or maximized \cite{2}.

We map the \(P\) model to the problem of selecting the optimal composition solution as follows:

- the search space \(S\) is represented by all the possible assignments of boolean values to semantic links in the graph. The boolean values determine if a certain link will be part of the solution.
- the set of constraints represents the necessary conditions for a selected part of the graph to be a solution, and determines the set \(S\) of feasible solutions.
- the function \(f\) represents the evaluation of a solution, which will take into consideration the \(QoS\) of the composing services and the degree of semantic matching between them.

Besides mapping the \(P\) model to Web service composition, the behavior of an ant should also be modeled. This behavior refers to the way an ant chooses among several candidate edges...
at a certain point. This choice will be done stochastically, with the probability for each of the edges depending on the level of pheromone on the edge and some other heuristic information such as the QoS of the service at the other end of the edge and the degree of semantic matching between the two services determining the edge. The probability \([2]\) to pick an edge \((i, j)\) having already constructed the partial solution \(s^p\) is as follows:

\[
P_{i,j}^k = \left\{ \begin{array}{ll}
\frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{e \in N(s^p)} \tau_{pq}^\alpha \eta_{pq}^\beta} & \text{if } e_{pq} \in N(s^p), \\
0 & \text{otherwise.}
\end{array} \right.
\]

where \(N(s^p)\) is the set of edges which may be added, and \(\eta_{ij}\) is the heuristic information associated with an edge. This score is computed as follows:

\[
\eta_{ij} = \frac{w_{QoS} \times QoS(S_2) + w_{Match} \times DoM(S_1, S_2)}{w_{QoS} + w_{Match}}
\]

where \(S_2\) is the service at the other end of the edge relative to the position of the ant. We compute the heuristic score by taking into account the quality of service of \(S_2\) and its degree of match \((DoM)\) with the service the ant is currently on, while also assigning weights to these components - \(w_{QoS}\) and \(w_{Match}\) respectively.

Another situation that needs to be modeled is represented by the solutions that will no longer have clusters as nodes, but individual services. Once we reach a node, we will immediately pick a service from that node. The decision of which service to pick will again be done probabilistically, this time associating probabilities to services proportional to their QoS. Therefore, the probability to pick a service \(s_i\) in a node is:

\[
p_{s_i} = \frac{QoS(s_i)}{\sum_{s_j \in C} QoS(s_j)}
\]

where \(C\) is the cluster \(s_i\) belongs to.

### B. The Selection Algorithm

Our selection algorithm adapts the Ant Colony Optimization (ACO) metaheuristic. Just as most ACO algorithms do, the selection algorithm iteratively searches for the best composition solutions. This section presents some theoretical background as well as the selection algorithm.

1) **Theoretical Background:** The selection algorithm extracts the best solutions from the composition graph. We define a solution as a directed acyclic graph having services as nodes (and containing the output node), such that for each input of each service there is another service in the solution providing that input via a directed edge. Within an iteration, multiple artificial ants traverse the composition graph to build solutions. In the first step, an ant is placed randomly on a node in the graph. The first thing an ant will do is try to find input concepts for the node it is currently on. To do this, it will go on a backward edge. When on another node, the ant will again try to find inputs for that node. This operation will be performed until there are no inputs that are not provided in the current partial solution. Once this is done, the ant will pick a forward edge from one of its nodes. It will try to move towards the output node, so it will try to go "forward", by picking edges going from nodes that are not providing inputs for any other nodes already 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 provided). We must be careful at each step so that the edge we add will not create a cycle in the graph, as we must be able to build a topological ordering of the services in the solution.

After the solution is constructed, it is evaluated, and if the score is high enough it is added to the list of best solutions found so far. The score of the solution is computed according to the following formula:

\[
Score(solution) = \frac{\sum_{e \in E_{solution}} \eta_e}{|E_{solution}|}
\]

One more aspect which needs to be addressed is how the level of the pheromone is updated during the execution of the algorithm. There are two types of pheromone updates which are performed, as suggested by the ACO metaheuristic. The first one is the local update which is performed by each ant at each step it makes on the graph, and the used formula \([2]\) is:

\[
\tau_{ij} = (1 - \varphi) \times \tau_{ij} + \varphi \times \tau_0
\]

where \(\tau_{ij}\) is the current pheromone level, \(\varphi\) is the so called decay coefficient, and \(\tau_0\) is the initial value of the pheromone.

The offline pheromone update is applied at the end of each iteration by only one ant, namely the one which has found the best solution in the current iteration. Such an update can be performed either on the iteration-best solution, or on the best-so-far solution. Since at the end of the algorithm we want to end up with more than one good solutions to be presented to the user, iteration-best is more suitable. The update formula \([2]\) is the following:

\[
\tau_{i,j} = (1 - \rho) \times \tau_{i,j} + \rho \times \Delta \tau_{ij}
\]

where \(\Delta \tau_{ij} = 1/\text{S}_{\text{best}}, \text{S}_{\text{best}}\) is the best score of a composition in the current iteration, as discussed above, if the edge belongs to \(\text{S}_{\text{best}},\) and 0 otherwise. \(\rho\) is the so called evaporation rate.

Throughout this section we have evaluated services according to their quality of service using the function \(QoS(s)\). This function is defined as follows:

\[
QoS(s) = \sum_{i} w_i \times Attr_i(s) \over \sum_{i} w_i
\]

where \(Attr_i(s)\) represents the value of the i-th quality of service attribute we consider, and \(w_i\) represents its weight in the final mean. Our QoS model considers the following quality of service attributes: response time, availability, throughput, successability, reliability, compliance, best practices, latency and documentation.
2) Pseudocode: The pseudocode of the main procedure which will be used is presented in Algorithm 3. In line 1, a new list which will hold the best compositions found while executing the algorithm. We do not go into complex implementation details here, but it can be mentioned that an appropriate data structure for this list would be a min-heap (because each time a good solution is found, the list member with the minimum score will need to be removed). On line 2, we initialize the variable which will count the number of iterations which have passed since a solution has been added to the list. We also initialize the set of ants to be used. Next, the main iteration loop is started. The algorithm will keep looping until a certain number of iterations have passed since an update has been made to the solution list. Each of the ants is then told to find a solution (line 8). Once this is done, all the solutions are considered for addition to the best solution list (lines 9 - 12). Also, the solution in the current iteration with the highest score is identified (lines 13 - 15). The offline pheromone update is applied on the edges of this solution (line 16). Finally, the list containing the best compositions is returned (line 17).

There are several procedures which are called in this algorithm. On line 8, the "FindSolution" function is invoked. It constructs a solution to the composition, and will be detailed shortly. The names of the procedures called on the composition list are self-explanatory and need no further discussion. The "score()" procedure computes the score of a solution according to the previously given formula. In line 16, the "ApplyOfflinePheromoneUpdate()" procedure is called. As its name states, it applies the offline pheromone update as presented earlier in this section.

We will now focus our attention on the procedure by which an ant builds a solution. Its pseudocode is shown in Algorithm 4. On the first line, an empty solution is initialized. Then a random node is chosen, and an appropriate service is chosen from the cluster it represents. The node is added to the partial solution. After that, we enter a while statement which will loop until a solution is found. In this loop, we first look to find all the required inputs for the partial solution. This is done by the "findInputs()" procedure, which we will detail in what follows. Then, if the output node is not yet present, we look to go forward and find another node for which the current solution can provide inputs. For this we use the "findNextService()" procedure, also to be detailed. Finally, after we exit the loop, the algorithm returns the found solution. There are a number of procedures used: "randomNode()" picks a random node of the graph, "pickService()" chooses a service from a cluster and "isSolution()" evaluates the current sub-graph to test if it is indeed a solution. These functions will not be detailed further.

We will now focus our attention on the procedure which finds inputs for the services in a partial solution. Its pseudocode is detailed in Algorithm 5. This procedure will be used recursively until all the inputs are provided. When considering a node, it first iterates through all the inputs which still don’t have providers. It then finds a providing node, such that by adding the corresponding edge the graph remains acyclic. As discussed before, this node is chosen stochastically, with the probabilities given by the pheromone level and heuristic score of the edges connecting the current node and the considered one. Then, the algorithm adds the node and the edge to the graph if they are not already present. Next, it assigns the provided concept for the searched input concept, and if the newly found node has not provided inputs itself, it calls the procedure recursively with the new node as parameter. In the last instruction, it returns the solution it has built so far.

The last procedure which will be discussed here is the one that picks new node to add to a partial solution with
The complexity of the algorithms presented in this section will now be assessed. In the worst case scenario, when constructing a solution the entire graph is traversed, so the complexity of Algorithm 4 is $O(V+E)$. Based on this, the complexity of Algorithm 3 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, 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.

V. Experimental Results

In order to evaluate our approach on a concrete composition, 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. In this subsection we will show the way the proposed composition technique behaves under this scenario, and then we evaluate the obtained results. For the considered scenario the user request is presented in Table I.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatientOntology.owl#PersonName</td>
<td>PatientOntology.owl#City</td>
</tr>
<tr>
<td>PatientOntology.owl#Address</td>
<td>PatientOntology.owl#Hospital</td>
</tr>
<tr>
<td>PatientOntology.owl#Country</td>
<td>PatientOntology.owl#Room</td>
</tr>
<tr>
<td>PatientOntology.owl#Symptom</td>
<td>PatientOntology.owl#TransportNumber</td>
</tr>
<tr>
<td>PatientOntology.owl#Physician</td>
<td>PatientOntology.owl#DateTime</td>
</tr>
</tbody>
</table>

An example of services involved in this scenario is presented in Table II and Table III. All the concepts in these services are part of a "PatientOnology".

<table>
<thead>
<tr>
<th>Service ID</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Symptom</td>
<td>Diagnosis</td>
</tr>
<tr>
<td>Cluster ID</td>
<td>5</td>
<td>Physician</td>
</tr>
<tr>
<td>QoS Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response Time: 314</td>
<td>Availability: 59</td>
<td>Throughput: 4</td>
</tr>
<tr>
<td>Successability: 96</td>
<td>Reliability: 67</td>
<td>Compliance: 84</td>
</tr>
<tr>
<td>Best practices: 85</td>
<td>Latency: 493</td>
<td>Documentation: 93</td>
</tr>
</tbody>
</table>
TABLE III
EXAMPLE OF WEB SERVICE 2

<table>
<thead>
<tr>
<th>Service ID</th>
<th>City</th>
<th>Hospital</th>
<th>Room</th>
<th>DateTime</th>
<th>TransportNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster ID</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>TransportNumber</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QoS Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Time: 1367</td>
</tr>
<tr>
<td>Availability: 74</td>
</tr>
<tr>
<td>Throughput: 6</td>
</tr>
<tr>
<td>Reliability: 69</td>
</tr>
<tr>
<td>Compliance: 76</td>
</tr>
<tr>
<td>Best practices: 76</td>
</tr>
<tr>
<td>Latency: 233</td>
</tr>
<tr>
<td>Documentation: 79</td>
</tr>
</tbody>
</table>

For the considered scenario we set the number of solutions we want to get (solution list size) to 5, the number of ants to 4 and the number of iterations in which no change has occurred in the best compositions list until we stop to 3. We also set all the weights according to which the score of a solution is computed 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.

The optimal composition solution identified has a score of 0.8459. The algorithm returned four 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 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: good solutions provided to the user with a small amount of computation required.

VI. CONCLUSION

In this paper, we have proposed an ant-inspired Web service composition technique. The proposed approach combines a service composition graph model with an ant colony optimization metaheuristic to find the optimal composition solution. The Web services involved in the service composition graph are provided by a service discovery process. To improve the efficiency and accuracy of services discovery process we have organized the set of available services into service clusters. We consider that a set of services belong to the same cluster if they provide similar functionality and there is a degree of match between their semantic descriptions. We have tested the composition technique on a set of scenarios from the medical domain.

REFERENCES