Algorithm for Automatic Web API Composition

Yong-Ju Lee

School of Computer Information, Kyungpook National University, 386 Gajangdong, Sangju, South Korea
yongju@knu.ac.kr

***Abstract*—Data mashup is a special class of mashup application that combines Web APIs from several data sources to generate a new and more valuable dataset. Although the data mashup has become very popular over the last few years, there are several challenging issues when combining a large number of APIs into the data mashup, especially when composite APIs are manually integrated by mashup developers. This paper proposes a novel algorithm for automatic composition of Web APIs. The proposed algorithm consists of constructing a directed similarity graph and searching composition candidates from the graph. We construct a directed similarity graph which presents the semantic functional dependency between the inputs and the outputs of Web APIs. We generate directed acyclic graphs (DAGs) that can produce the output satisfying the desired goal. We rapidly prune APIs that are guaranteed not to involve the composition in order to produce the DAGs efficiently. The algorithm is evaluated using a collection of REST and SOAP APIs extracted from ProgrammableWeb.**

Keywords-automatic composition algorithm; semantic data mashup; ontology learning method; Web API

I. INTRODUCTION

A mashup is a Web application that combines data, presentation, or functionality from several different sources to create new services. An example of the mashup is HousingMaps [1], which displays available houses in an area by combining listings from Craigslist with a display map from Google. A *data mashup* is a special class of the mashup application that combines data from several data sources (typically provided through Web APIs; these API types are usually SOAP, REST, JavaScript, XML-RPC, Atom, etc.) to generate a more meaningful dataset. Data mashups have become very popular over the last few years. For example, as of August 2012, ProgrammableWeb [2] has published more than 7000 Web APIs. Several mashup tools such as Yahoo's Pipes, IBM's Damia, and Intel's Mashmaker have been developed to enable users to create data mashups without programming knowledge.

Although the data mashup has emerged as a common technology for combining Web APIs, there are several challenging issues. First, since a portal site may have a large number of APIs available for data mashups, manually searching and composing compatible APIs can be a tedious and time-consuming task. Therefore, mashup developers wish to quickly find the desired APIs and easily integrate them without having to expend considerable programming efforts. Second, portal sites typically only support keyword search or category search. These search methods are insufficient due to their bad recall and bad precision. To make mashups more efficiently, we need a semantic-based approach such that agents can reason about the capabilities of the APIs that permit their discovery and composition. Third, most mashup developers want to figure out all the intermediate steps needed to generate the desired mashup automatically. An infrastructure that allows users to provide some interesting or relevant composition candidates that can possibly incorporate with existing mashups is needed.

To solve the above issues, we present an algorithm for automatic discovery and composition of Web APIs using their semantic descriptions. Given a formal description of the Web API, a desired goal can be directed matched to the output of a single API. This task is called *discovery*. If the API is not found, the agent can search for two or more APIs that can be composed to satisfy the required goal. This task is called *composition*. Since the discovery is a special case of the composition where the number of APIs involved in the composition is exactly equal to one, discovery and composition can be viewed as a single problem.

We define API descriptions to syntactically describe Web APIs, and use an ontology learning method [3] to semantically describe Web APIs. We propose a Web API composition algorithm based on the ontology learning method. The proposed algorithm consists of constructing a directed similarity graph and searching composition candidates. The composition process can be described as that of generating directed acyclic graphs (DAGs) that can produce the output satisfying the desired goal, where the DAGs are gradually generated by forward-backward chaining of APIs. In order to produce the DAGs efficiently, we filter out APIs that are not useful for the composition. The main contributions from this paper are as follows:

- The paper proposes a new efficient algorithm for solving the Web API composition problem that takes semantics into account. The proposed algorithm automatically selects the individual APIs involved in the composition for a given query, without the need for manual intervention.
- Selecting and integrating APIs suitable for data mashups are critical for any mashup toolkits. We show in this paper how the characteristics of APIs can be syntactically defined and semantically described, and how to use the syntactic and semantic descriptions to aid the easy discovery and composition of Web APIs.
- A semantic-based data mashup tool is implemented for lowering the complexity of underlying programming efforts. Using this tool, the composition of APIs does not require in-depth programming knowledge. Users are able to integrate APIs with minimal training.

 ISBN: 978-1-61208-248-6

The rest of this paper is organized as follows. Section 2 begins by introducing our ontology learning method. Section 3 describes automatic Web API discovery and composition algorithms. Section 4 describes an implementation and experiment. Section 5 discusses related work, and Section 6 contains conclusions and future work.

II. ONTOLOGY LEARNING METHOD

The successful employment of semantic Web APIs is dependent on the availability of high-quality ontologies. The construction of such ontologies is difficult and costly, thus hampering Web API deployment. Our ontology learning method [3] automatically generates ontologies from Web API descriptions and their underlying semantics.

A. Parameter Clustering Technique

We have developed a parameter clustering technique to derive several *semantically meaningful concepts* from API parameters. We consider the syntactic information that resides in the API descriptions, and apply a mining algorithm to obtain their underlying semantics. The main idea is to measure the co-occurrence of terms and cluster the terms into a set of concepts. Formally, we can define an API as follows:

Definition 1: A Web API W=<I, O> where I is the input and O is the output. Each input and output contains a set of parameters for the API.

The input/output parameters are often combined as a sequence of several terms. We utilize a heuristic as the basis of our clustering, in that the terms tend to express the same concept if they frequently occur together. This allows us to cluster terms by exploiting the conditional probability of their occurrences in the input and output of Web APIs, specifically we are interested in the *association rules* [4]. We use the agglomerative hierarchical clustering algorithm to turn the set of terms $T = \{t_1, t_2, \ldots t_m\}$ into the concepts $C = \{c_1, c_2, \ldots c_n\}$. For example, the terms {`zip`, `city`, `area`, `state`} can be treated as one concept, they are grouped into one cluster.

B. Pattern Analysis Technique

The pattern analysis technique captures *relationships between the terms* contained in a parameter, and matches the parameters if both terms are similar and the relationships are equivalent. This approach is derived from the observation that people employ similar patterns when composing a parameter out of multiple terms. Based on the experimental observations, the relationships between the terms are defined in Table 1. Two ontological concepts are matched if and only if one of the following is true; (1) one concept is a *property* of the other concept, and (2) one concept is a *subclass* of the other concept.

From the above rules, an agent would be able to find a match based on the similarities of the API. For example, assume that a parameter `CityName` was to be compared against another parameter `CodeOfCity`. The keyword search would not count these as a possible match. However, if the `City` term had the relationships "X **propertyOf** Y" in its pattern rule, the matching logic will return a matching score because these two parameters are closely related (perhaps using the rules "`CityName` **propertyOf** `City`" and "`CodeOfCity` **propertyOf** `City`").

TABLE I. RELATIONSHIPS BETWEEN TERMS

No	Pattern	Relationships
1	$Noun_1$+$Noun_2$	Parameter **propertyOf** $Noun_1$
2	Adjective+Noun	Parameter **subClassOf** Noun
3	Verb+Noun	Parameter **subClassOf** Noun
4	$Noun_1$+$Noun_2$+$Noun_3$	Parameter **propertyOf** $Noun_1$
5	$Noun_1$+Preposition+$Noun_2$	Parameter **propertyOf** $Noun_2$

C. Semantic Matching Technique

The semantic matching technique estimates the similarity of the input and output by considering the underlying concepts the input/output parameters cover. Formally, we describe the input as a vector $I =< p_i, C_i >$ (similarly, the output can be represented in the form $O =< p_o, C_o >$), where p_i is the set of input parameters and C_i is the concept that is associated with p_i. Then, the similarity of the input can be found using the following two steps (the output can be processed in a similar fashion); (1) we split p_i into a set of terms, we then find synonyms for these terms, and (2) we replace each term with its corresponding concepts, and then compute a similarity score.

The similarity score is defined to select the best matches for the given input. Consider a pair of candidate parameters p_i and p_j, the similarity between p_i and p_j is given by the following formula:

$$\mathrm{Sim}(p_i, p_j) = \frac{2 \times \left\| \mathrm{Match}(p_i, p_j) \right\|}{m+n}$$

where m and n denote the number of valid terms in parameters, $\left\| \mathrm{Match}(p_i, p_j) \right\|$ returns the number of matching terms. Here, the similarity of each parameter is calculated by the best matching parameter that has a larger number of semantically related terms. The overall similarity is computed by a linear combination [3] to combine the similarity of each parameter.

Since existing matching techniques based on the clustering consider all terms in a cluster as an equivalent concept and ignore any hierarchical relationships between the terms, matches might exist that are irrelevant to the user's intention (i.e., false positives). Thus, a pruning process is necessary to improve the precision of the results. The basic idea is to improve the precision of the matching technique by applying the pattern relationships defined in Table 1. For details, readers may refer to our previous work [3].

III. WEB API DISCOVERY AND COMPOSITION

A. Discovery Problem

Given a query and a collection of APIs stored in the registry, automatically finding an API from the registry that

 ISBN: 978-1-61208-248-6

matches the query requirement is the Web API discovery problem. For example, we are looking for an API to search a hotel. Table 2 shows the input/output parameters of a query and an API. In this example a Web API W satisfy the query Q. Q requires `HotelName` as the output and W produces `HotelName` and `ConfirmNumber`. The extra output produced can be ignored. W requires `CountryCode` and `NameOfCity` as the input and Q provides `CountryID`, `StateName`, and `CityName` as the input. An API parameter can be matched with the other parameter only if there is a semantic relationship between them. Here, although `CountryCode` and `CountryID` are different forms, they have the same semantics since they are referred to the same concept. Also `NameOfCity` and `CityName` have the same semantics since they are properties of the same object (i.e., `City`). Therefore, the agent is able to infer that Q and W input parameters have semantically the same classes.

TABLE II. EXAMPLE FOR DISCOVERY PROBLEM

API	Input Parameters	Output Parameters
Q	`CountryID, StateName, CityName`	`HotelName`
W	`CountryCode, NameOfCity`	`HotelName, ConfirmNumber`

We describe an *automatic Web API discovery algorithm* similar to the one in [5]. An API matches a query when an API is sufficiently similar to the query. This means that we need to allow the agent to perform matches that recognize the degree of similarity between APIs and the query. We define the matching criteria as follows:

Definition 2: An API W matches a query Q when all the output parameters of Q are matched by the output parameters of W, and all the input parameters of W are matched by the input parameters of Q.

Definition 2 guarantees that the API found satisfies the needs of the query, and the query provides all the input parameters that the API needs to operate correctly. Our discovery algorithm is shown in Algorithm 1. This algorithm adopts strategies that rapidly prune APIs that are guaranteed not to match the query, thus improving the efficiency of the system. A query is matched against all APIs stored in the registry. A match between a query and an API consists of matching all the output parameters of the query against the output parameters of the API; and all the input parameters of the API against the input parameters of the query. If one of the query's output parameters is not matched by any of the API's output, the match fails. Matching between inputs is computed by the same process, but with the order of the query and API reversed. The similarity score of a match between two parameters is calculated by the semantic matching technique described in the previous section. The APIs are returned in the descending order of similarity scores.

Algorithm 1: Discovery Algorithm
```
//input: query (Q), APIs
//output: matched APIs
for all APIs
  if Matching(Q, API) then result.append(API)
return Sort(result)
Matching(Q, API)
    SemanticMatch(Q.O, API.O)
    SemanticMatch(API.I, Q.I)
```

B. Composition Problem

Given a query and a collection of APIs, in case a matching API is not found, searching a sequence of APIs that can be composed together is the composition problem of Web APIs. It means that the output generated by one API can be accepted as the input of another API. For example, we are looking for APIs to find a hotel's location. Table 3 shows the input/output parameters of a query Q, and two Web APIs W_1 and W_2 in the registry. Suppose the agent cannot find a single API that matches the criteria, then it composes *n* APIs from the set of Web APIs available in the registry. In this table, W_1 returns `HotelName` as the output. W_2 receives it as the input and returns `Location` as the result. So, the subsequent W_2 may use the output produced by the preceding W_1 as the input.

TABLE III. EXAMPLE FOR COMPOSITION PROBLEM

API	Input Parameters	Output Parameters
Q	`CountryID, StateName, CityName`	`Location`
W_1	`CountryCode, NameOfCity`	`ConfirmNumber, HotelName`
W_2	`HotelName`	`Location`

Now we can define the Web API composition problem as follows:

Definition 3: If an API W_1 can produce O_1 as its output parameters and an API W_2 can consume O_1 as input parameters, we can conclude that W_1 and W_2 are *composable*. Then, the Web API composition problem can be defined as automatically finding a DAG of APIs from the registry.

We describe a Web API as <W.I, W.O> and a query as <Q.I, Q.O>. A composition is valid if the following conditions are satisfied:

1) $_{\exists}W_i\ (Q.I \supseteq W_i.I)$
2) $_{\exists}W_j\ (Q.O \subseteq W_j.O)$
3) $_{\forall}W_i, W_j$, there exists at least a path from W_i to W_j.

In other words, the APIs in the first stage of the composition can only use the query input parameters. The outputs produced by the APIs in the last stage of the composition should contain all the output parameters that the query requires to be produced. The output from an API at any

 ISBN: 978-1-61208-248-6

stage in the composition should be able to provide as the input to the next API.

The composition problem is just achieving a desired goal from the initial request, while not making it know the underlying composition details. The mashup developers can now simply describe a goal in form of the query, and submit the requirement to our system. If the desired goal can be directly matched to the output of a single Web API, the composition problem reduces to the discovery problem. Otherwise, it can be accomplished by searching a sequence of APIs that can produce the desired output. Such sequence composition of APIs can be viewed as a searching DAG that can be constructed from an initially given query. In particular, when all nodes in the graph have not more than one incoming edge and not more than one outgoing edge, the problem reduces to a linearly linked APIs problem. Because the discovery problem is a simple case of the composition where the number of APIs involved in the composition is exactly equal to one, the discovery and composition can be viewed as a single problem.

C. Constructing Directed Similarity Graph

In order to speed up the calculation of possible composition plans, we use a pre-computed directed similarity graph that chains the output of one API into the input of another API. The connection of the nodes is based on the semantic similarity between the output and input of the nodes. Algorithm 2 illustrates the construction procedure for the graph. At the beginning, we assign each API in the registry to vertexes iteratively. We then establish edges between the vertexes. For each vertex v_i, we check whether its corresponding output can be accepted as an input by a v_j by computing the similarity score. If the output of v_i is semantically similar to the input of v_j (i.e., $\mathrm{Sim}(v_i.O, v_j.I) > 0$), then we add a directed edge from v_i to v_j (in the reverse direction) and assign a similarity score. We also check if there exists a vertex v_j, whose output can be consumed by v_i as an input, in the similar manner. After constructing the directed similarity graph, we solve the composition problem within this graph. This initial graph is dynamically modified if new APIs become available.

Algorithm 2: Graph Construction Algorithm
```
//input: APIs
//output: a directed similarity graph
for all APIs
    v_i = addVertex(API)
for each v_i ∈ V
  for each v_j ∈ V
    if Sim(v_i.O, v_j.I)>0 then addEdge(v_i, v_j)
    if Sim(v_i.I, v_j.O)>0 then addEdge(v_i, v_j)
```

D. Graph-based Composition Algorithm

Our graph-based composition algorithm can be described as that of generating DAGs that can produce the output satisfying the desired goal. In order to produce the DAGs efficiently, we rapidly filter out APIs that are not useful for the composition. We extend our discovery algorithm to handle the composition problem. The algorithm is based on a modified Breath-First Search (BFS) algorithm [6] which can find a shortest path from a source vertex to a target vertex. We solve the composition problem in four main stages: searching sub-graphs, adding start nodes, validating candidates, and ranking candidates.

Searching sub-graphs: First, we search the API registry about any API that has all the output parameters of the query (we call "last nodes"), and any API that has at least one of the input parameters of the query (we call "first nodes"). After this searching it is assumed that non empty sets are obtained for the first and last nodes. The next is to create *n*-ary trees for every last node by visiting all the nodes connected to a particular last node. Such tree is constructed by recursively including nodes and edges from the directed similarity graph until we reach the first nodes. We use the BFS algorithm to solve this problem. Now we can find all the possible composition candidates from the trees. Figure 1 shows a general overview of the query and the matching APIs before constructing the overall composition plans.

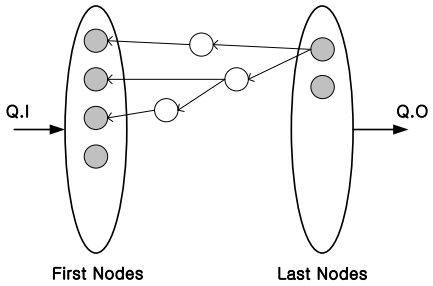


Figure 1. General Overview of Query and Matching APIs

Adding start nodes: In this stage, a start node is added to each of the trees. The start node is a special dummy node for a dynamically created API, namely the API that provides the input of the query. The start node is represented as $W_0 = <\emptyset, Q.I>$, namely W_0 is an API in a tree with no input, having only an output. Finding a possible composition candidate consists in generating a DAG from the start node to the last node in the trees. When a possible composition candidate has been found, all the nodes participating in the composition should be validated in the next stage.

Validating candidates: A possible composition candidate is valid if all nodes in the composition can be executed (non-)sequentially in order to produce the desired results. This validating is done by starting from the start node working our way backwards. At this point, first nodes consist of all the APIs such that all their inputs are provided by the start node. Let O_1 be a union of all outputs produced from the first nodes in the composition, and I_1 (i.e., Q.I) be the query input. Inputs for the second nodes are all the outputs

 ISBN: 978-1-61208-248-6

produced by the previous nodes and the query input, i.e., $I_2 = O_1 \cup I_1$. The combination I_2 will be the available input for the next nodes. This transition (i.e., $I_{i+1} = O_i \cup I_i$) is repeated until the last node is reached, removing redundant nodes which do not contribute to the optimal path at each step.

Ranking candidates: A DAG is considered as a composition candidate only if it meets the requirements of the output and input described in the query. It means all output parameters of the query must be obtained, and partly or fully the input parameters of the query must be consumed. After a composition candidate has been found, we gather all the similarity data from the edges involved in the composition in order to compute a similarity score. This score is calculated by the average value of all the similarity data related to the edges, and the ranking of the composition candidate is determined by the score. The list of composition candidates is ordered according to this ranking score and the head of the list is considered the best, recommended option for the user. Algorithm 3 illustrates our graph-based composition algorithm.

Algorithm 3: Composition Algorithm
//input: query (Q), a directed similarity graph
//output: ranked composition candidates
if SemanticMatch(Q.O, API.O) is empty **then** fail
if SemanticMatch(API.I, Q.I) is empty **then** fail
for each last node
Call BFS algorithm
Create *n*-ary trees
for each tree
Adding a start node to the tree
Generating a DAG from start node to last node
//Validating possible composition candidates
i = 1, I_i = Q.I
L_i = NextApiList(i)
while Not (last node $\wedge$ $V_i \equiv \emptyset$)
O_i = UnionAllOutputs(L_i)
$I_{i+1} = O_i \cup I_i$
L_{i+1} = NextApiList(i+1)
Removing redundant nodes
i = i+1
endwhile
endfor
Ranking composition candidates

IV. IMPLEMENTATION AND EXPERIMENT

We developed a semantic-based data mashup tool. The system architecture is shown in Figure 2. The composition planner is responsible for planning to achieve the composition relevant to the desired goal. It captures the current composition states and dynamically composes relevant APIs that can be added to the mashup. The mashup engine interprets the composition of corresponding APIs and displays the immediate results. In the graphical user interface (GUI), mashup developers can obtain the immediate composition results visually, and iteratively refine their goals until the final results satisfying. The ontology learning method automatically builds semantic ontologies from Web API descriptions.

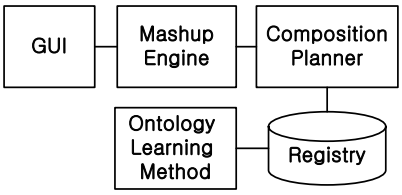


Figure 2. System Architecture

To experiment with the data mashup tool we extracted a collection of REST and SOAP APIs from ProgrammableWeb. To avoid potential bias, we chose different APIs from different domains. We first collected a subset which associated REST APIs for three domains: weather, travel, and mapping. This set contains 63 APIs. Next, we collected a subset containing 17 SOAP APIs from three domains: zipcode, location, and search. In Figure 3, we show a directed similarity graph which obtained from our experimental dataset. The graph consists of 80 nodes and 123 edges.

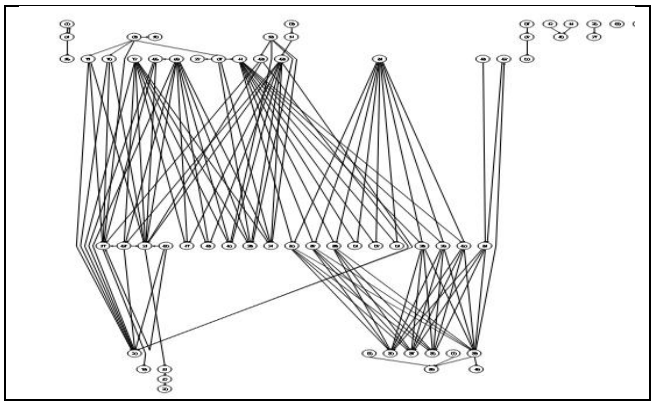


Figure 3. Directed Similarity Graph

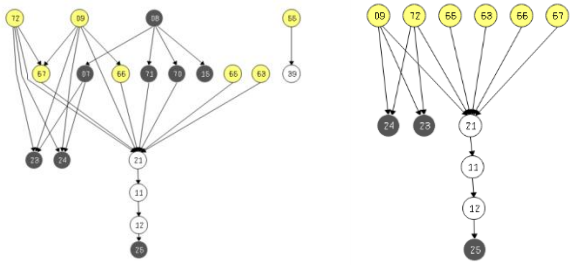
A possible query for the Web API composition is given as follows: Q.I = {`zipcode`}, Q.O = {`city`, `latitude`, `longitude`}. The composition result is exemplified by part of the directed similarity graph as shown in Figure 4. From the registry our engine has discovered 8 last nodes (dark grey circles) and 7 first nodes (light grey circles). We call the BFS algorithm and create an *n*-ary tree for each last node. This is repeated until all the last nodes are reached.

A total of 3 possible composition candidates have been automatically generated from the graph. As we have mentioned in Section 3.D, a start node W_0 is added to each tree and the validation of candidates is performed for optimal paths. After running the validation, final composition candidates are selected and similarity scores are calculated. In Table 4, we list these ranked composition candidates.

To evaluate our composition quality, we check how many of desired goals are captured by the composition algorithm. We can observe that two third of all the recommen-

 ISBN: 978-1-61208-248-6

ded results in Table 4 have desired or relevant goals. Although the 3rd ranking result turns out to be invalid as it does not satisfy the user requirement, top 2 ranking results have desired composition plans. These results have shown that our algorithm can generate most user desired outputs.



(a) Discovered last and first nodes (b) *n*-ary trees

Figure 4. Result of Graph-Based Composition Algorithm

TABLE IV. LIST OF RANKED COMPOSITION CANDIDATES

Rank	Score	DAG
1	0.625	W_0→(9, 72) →23
2	0.550	W_0→ (9, 72) →24
3	0.222	W_0→ (9, 72, 65, 66, 67) →21→11→12→25

V. RELATED WORK

Most researches handing the automatic composition problem have been focusing on the composition of SOAP-based Web services. Many various techniques have been used for this study, such as graph-based search algorithm [7] and AI planning [8]. However, the work presented in this paper is not limited to composing SOAP-based Web services, but also considers REST, JavaScript, XML-RPC, and Atom Web APIs.

The use of graph-based search algorithms to solve the composition problem has been studied before. Kona et al. [7] propose an automatic composition algorithm for semantic Web services. Rodriguez-Mier et al. [9] propose a heuristic-based search algorithm for automatic Web service composition. Shiaa et al. [10] present an incremental graph-based approach to automatic service composition. These works are similar to our study. However, they cannot find an optimal solution, and do not support various Web API protocols.

We recently proposed an automatic Web API composition algorithm [11] to handle the sequential composition problem. This paper is an extension of our previous work and focuses on the (non-)sequential composition that can be represented in the form of directed acyclic graphs (DAGs). This is the most general case of the Web API composition.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents an algorithm for the automatic Web API composition. This algorithm is based on the graph-based approach, where composition candidates are gradually generated by forward-backward chaining of APIs. Our algorithm can get optimal plans by applying strategies that rapidly prune APIs that are guaranteed not to match the query. A key issue is how to locate the desired APIs. The efficient discovery can play a crucial role in conducting further API composition. We define API descriptions that syntactically describe Web APIs, and use an ontology learning method that semantically describes APIs. These syntactic and semantic descriptions allow the agent to automate the composition of Web APIs.

Our future work is focusing on the investigation of the performance and scalability measures for the proposed graph-based composition algorithm. By this we aim to optimize the functionality of our system. We are also exploring various optimization techniques that can apply to the algorithm. For example, a heuristic AI planning technique can be used to find an optimized solution with a minimal number of paths. The use of dynamic optimization techniques over the graph helps greatly in obtaining the effectiveness and efficiency of our approach.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (No. 2010-0008303).

REFERENCES

[1] http://www.housingmaps.com

[2] http://www.programmableweb.com

[3] Y. J. Lee and J. H. Kim, "Semantically Enabled Data Mashups using Ontology Learning Method for Web APIs," Proceedings of the 2012 Computing, Communications and Applications Conference, 2012.

[4] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," Proceedings of the 1993 ACM-SIGMOD International Conference Management of Data, 1993.

[5] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic Matching of Web Services Capabilites," Proceedings of the International Semantic Web Conference (ISWC), 2002.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms (Second Edition), MIT Press, 2001.

[7] K. Kona, A. Bansal, M. Blake, and G. Gupta, "Generalized Semantics-based Service Composition," Proceedings of the IEEE International Conference on Web Services (ICWS), 2008.

[8] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN Planning for Web Service Composition using SHOP2," Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 1, No. 4, pp. 377-396, 2004.

[9] P. Rodriguez-Mier, M. Mucientes, and M. Lama, "Automatic Web Service Composition with a Heuristic-based Search Algorithm," Proceedings of the International Semantic Web Conference (ISWC), 2011.

[10] M. Shiaa, J. Fladmark, and B. Thiell, "An Incremental Graph-based Approach to Automatic Service Composition," Proceedings of the International Semantic Web Conference (ISWC), 2008.

[11] Y. J. Lee and J. S. Kim, "Automatic Web API Composition for Semantic Data Mashups," Proceedings of the 4th International Conference on Computational Intelligence and Communication Networks (CICN), 2012.

 ISBN: 978-1-61208-248-6