Semantic-Based Web API Composition for Data Mashups*

YONG-JU LEE
School of Computer Information, Kyungpook National University, 386 Gajangdong, Sangju, South Korea

With the growing popularity of data mashups, the number of Web APIs has increased significantly. As a result, finding and composing the right APIs has become an increasingly complex task. Although several tools such as Yahoo’s Pipes, IBM’s Lotus Mashup, and Intel’s Mashmaker have been developed to enable users to create data mashups without programming skills, there are several challenging issues when combining a large number of APIs into the data mashup. This paper proposes novel algorithms for the automatic discovery and composition of Web APIs. Our discovery algorithm adopts strategies that rapidly prune APIs that are guaranteed not to match the query. Our composition algorithm consists of constructing a composable similarity graph (CSG) and searching composition candidates. The CSG presents the semantic functional dependency between the inputs and the outputs of the Web APIs. Using this graph, we generate directed acyclic graphs (DAGs) that can produce the output satisfying the desired goal. We evaluate the algorithms on a real-world dataset from ProgrammableWeb.com, and show that they can produce the results satisfying the user’s desired output.

Keywords: data mashup, Web API, discovery, composition, ontology learning, graph-based algorithm

1. INTRODUCTION

A mashup is a Web application that combines data, presentation, or functionality from several different sources to create new services. A data mashup is a special class of the mashup application that combines data from several data sources to generate a more meaningful dataset. Typically these data sources are provided through Web APIs, these APIs include SOAP, REST, JavaScript, and XML-RPC protocols. The growing interest in data mashups has encouraged the development of a large number of Web APIs. For example, as of November 2014, ProgrammableWeb [1] has published more than 12,000 Web APIs. There are also several tools to build data mashups without programming knowledge, such as Yahoo’s Pipes, IBM’s Lotus Mashup, and Intel’s Mashmaker. Although these tools provide valid solutions that make it easier to compose new APIs, they can only interact with a limited number of APIs, generally those that deal with the internal APIs of the company that developed the tool.

Recent approaches have stressed the importance of Web API composition [2], data mediation [3], and the semantic annotations of Web APIs [4]. Although they try to overcome the limitations of traditional data mashup solutions, there are several challenging issues. First, because a registry may have a large number of APIs available for data mashups, manually searching and composing compatible APIs can be a tedious and

Communicated by Yong-Ju Lee.

*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).
time-consuming task. Therefore, developers wish to quickly find the desired APIs and easily integrate them without having to expend considerable programming efforts. Second, on-line portal sites typically only support keyword or category search. The keyword search is insufficient because of bad recall and precision. Returned lists from the category search are generally based on criteria that have no relevance to the developers desired goals (e.g., in alphabetical or last update time order). To create data mashups more efficiently, a semantic-based approach is needed 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 [5]. An infrastructure that allows users to provide interesting or relevant composition candidates that can be possibly incorporated with the existing mashups is needed.

To address the above issues, we present automatic Web API discovery and composition algorithms. Given a formal description of the Web API, a desired goal can be directly 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. Because discovery is a special case of composition, where the number of selected APIs involved in the composition is equal to one, discovery and composition can be viewed as a single problem.

We define API descriptions to syntactically describe Web APIs and introduce an ontology learning method [6] to semantically describe them. This paper proposes algorithms for automatic Web API discovery and composition using the ontology learning method. A common issue is how to locate the desired APIs. Efficient discovery can play a critical role in conducting further API composition. Our discovery algorithm adopts strategies that rapidly filter out APIs that are guaranteed not to match the query. Our composition algorithm consists of constructing a composable similarity graph (CSG) and searching composition candidates. The composition process can be described as generating directed acyclic graphs (DAGs) that can produce the output satisfying the desired goal. The DAGs are gradually generated by forward and backward searching over the graph. The main contributions of this paper are as follows:

- We propose an efficient algorithm for solving the Web APIs composition problem that considers semantics. The proposed algorithm automatically selects the individual APIs involved in the composition for a given query, without human intervention.
- Selecting and composing 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. We also indicate how to use the syntactic and semantic descriptions to aid the automatic discovery and composition of Web APIs.
- A semantic-based Web APIs composition engine is implemented for reducing the complexity of the underlying programming efforts. Using this engine, the composition of Web APIs does not require in-depth programming knowledge. Users are able to use and integrate APIs with minimal training.

The remainder of this paper is organized as follows. Section 2 describes related work. The ontology learning method is introduced in Section 3. Section 4 presents the automatic Web API discovery and composition algorithms. Section 5 describes the ex-
perimental evaluation. The conclusions and future work are described in Section 6.

2. RELATED WORK

A large number of Web APIs raises a challenging issue of how to locate a desired API. The traditional keyword searching method is inaccurate and its limitations have been noted for several years. Adding semantics to Web APIs may help overcome these limitations. A majority of these approaches [3, 4, 7] have semantic annotations such as SMCDL (Semantic Mashup Component Description Language) or MCR (Mashup Component Repository). However, building semantic annotations is a relatively time-consuming and labor-intensive task. A number of ontology learning methods have been proposed for the automatic acquisition of semantic annotations including Naïve Bayes [8], clustering [9], automatic extracting [10], and ontological relationships [11]. However, these researches have focused on SOAP-based Web services.

Several techniques have been used for the automatic discovery and composition problem such as AI planning [12] and graph-based searching [13]. AI planning techniques, however, have some drawbacks including high complexity, expensive computation, and inferior performance. Graph-based searching techniques are easier than AI planning techniques. However, most of these approaches rely on very complex dependency graphs that have not been optimized to reduce data redundancy. They do not support the different Web API protocols such as SOAP, REST, JavaScript, and XML-RPC. Our paper presents a graph-based Web API composition algorithm to address the automatic composition problem. The use of graph-based composition algorithms to solve the composition problem has been studied previously.

Shiaa et al. [14] presented an incremental graph-based approach to automatic service composition. This approach relies on semantic annotations on inputs, outputs, goals, and non-functional properties which are created manually. It does not take into account the use of filtering techniques in order to speed-up the search, so searching for an optimal composition in a large registry may be infeasible. Kona et al. [13] proposed an automatic composition algorithm for semantic Web services. The graph is calculated iteratively starting with the input parameters provided by the requester. Although the useless services are filtered, the algorithm cannot find an optimal composition. A forward and backward searching method over the graph is required in order to minimize the number of services in the composition. Rodriguez-Mier et al. [15] proposed a heuristic-based search algorithm for automatic Web service composition. The use of the A* algorithm allows finding an optimal composition with a minimal number of services and execution path. But a main drawback of this heuristic method is the large search space. A* algorithm is space-limited in practice and is no more practical than Breath-First Search (BFS) algorithm.

Our approach uses BFS algorithm [16] to find a minimal composition that satisfies the user request. The composition is performed by forward and backward searching over the graph, which maximizes the parallel execution of Web APIs. We recently proposed an automatic Web API composition algorithm [17] to address 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 DAGs. This is the
most general case of the Web API composition. Furthermore, an experimental study for the proposed algorithm has been done on a real-world dataset from ProgrammableWeb.

3. ONTOLOGY LEARNING METHOD

Web APIs accept programmatic inputs and produce programmatic outputs [18]. Although programmatic inputs/outputs provide an easy and intuitive method of using APIs, their limitations must also be considered. As input/output parameters are freely and arbitrarily chosen, instead of relying on a controlled vocabulary, parameter ambiguity will likely cause a mismatch between APIs. Furthermore, parameters only represent a flat structure with no hierarchy, thus a large number of parameters could cause difficulties for developers in matching compatible APIs. The solution to automatic Web API composition presented here is based on the ontology learning method, whose principles and techniques have been presented in [6]. In this section, we provide a brief introduction of the method.

3.1 Parameter Hierarchical Clustering

We have developed a hierarchical clustering technique to derive several semantically meaningful concepts from the 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 then cluster the terms into a set of concepts. Formally, we can define an API as follows:

**Definition 1**: A Web API $W = \langle I, O \rangle$ 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. For example, in the parameter `ArrivalTimeOfAirplane`, the terms are specified by their first letter capitalized `{Arrival, Time, Of, Airplane}`. When clustering terms residing in the parameters into several meaningful semantic concepts, we consider the co-occurrence of terms. A common heuristic is that the terms tend to express the same concept if they frequently occur together. This allows us to cluster the terms by exploiting the conditional probability of their occurrence in the inputs and outputs of the APIs.

Let $T = \{t_1, t_2, \ldots, t_m\}$ be a set of terms. Let $IO$ be a set of candidate inputs/outputs available from the API descriptions. To reflect co-occurrence, we introduce association rules [19] of the form $t_i \rightarrow t_j$, where both $t_i$ and $t_j \in T$. The rule $t_i \rightarrow t_j$ holds in the inputs/outputs set $IO$ with support and confidence. The support $s$ is the probability that $t_i$ occurs in an input/output. The confidence $c$ is the probability that $t_j$ occurs in an input/output, given that $t_i$ is known to occur in the input/output. The problem of association rules can be computed using the well-known Apriori algorithm [20].

Ideally, parameter-clustering results should have the following two features: the cohesion within a concept (connections between the parameters inside the concept) should be strong and the correlation between the concepts (connections between the parameters in different concepts) should be weak. We say that $t_i$ is closely associated to $t_j$ if the
confidence of the rule \( t_i \rightarrow t_j \) is greater than a threshold value \( \delta \) (i.e., \( c(t_i \rightarrow t_j) > \delta \)). To measure the overall quality of clustering, we define \( score = coh/cor \) where \( coh \) and \( cor \) are the average of all the cohesion and correlation values, respectively. Our goal is to obtain a high score that will reflect tight connections inside the clusters but loose connections between clusters. We use the agglomerative hierarchical clustering algorithm [21] to turn the set of terms \( T = \{t_1, t_2, \ldots, t_m\} \) into the concepts \( C = \{c_1, c_2, \ldots, c_n\} \).

3.2 Parameter Pattern Analysis

The parameter 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. The distribution of Web APIs in ProgrammableWeb is that almost two third of the APIs (72%) are REST, 18% of the APIs are SOAP, 6% are JavaScript, and 3% are XML-RPC. We adopt REST and SOAP as two major classes of Web APIs, because we consider SOAP, JavaScript, and XML-RPC to be a family of RPC-style APIs. In order to characterize the patterns, therefore, 8,209 parameters from 168 REST APIs and 1,245 parameters from 50 SOAP APIs collected from the Internet were categorized into buckets.

As shown in Table 1, 2,435 (30%), 752 (9%), 608 (7%), 472 (6%), and 368 (5%) parameters in the REST APIs were defined as the noun phrases \( Noun_1+Noun_2 \), \( Adjective+Noun \), \( Verb+Noun \), \( Noun_1+Noun_2+Noun_3 \), and \( Noun_1+Preposition+Noun_2 \), respectively. There were 3,574 (43%) parameters not covered by any of the patterns. The majority of them (3,548 parameters) contained only one token (e.g., City). The other 26 parameters could not be tokenized according to the patterns defined in Table 1. This table also shows the pattern analysis results with respect to the SOAP APIs. 470 (37%), 175 (14%), 70 (6%), 40 (3%), and 15 (1%) parameters in the SOAP APIs were defined as the noun phrases \( Noun_1+Noun_2 \), \( Noun_1+Noun_2+Noun_3 \), \( Verb+Noun \), \( Noun_1+Preposition+Noun_2 \), and \( Adjective+Noun \), respectively. There were 499 (39%) parameters not covered by the patterns.

<table>
<thead>
<tr>
<th>Table 1. Pattern Analysis for REST and SOAP APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REST APIs</strong></td>
</tr>
<tr>
<td>( Noun_1+Noun_2 )</td>
</tr>
<tr>
<td>( Adjective+Noun )</td>
</tr>
<tr>
<td>( Verb+Noun )</td>
</tr>
<tr>
<td>( Noun_1+Noun_2+Noun_3 )</td>
</tr>
<tr>
<td>( Noun_1+Preposition+Noun_2 )</td>
</tr>
</tbody>
</table>

We can observe that there are five common patterns for REST and SOAP APIs, although the rate of their occurrence is different depending on the protocol. We capture the relationships between the terms in a parameter. From the above table, our pattern analysis rules are defined in Table 2.
Table 2. Relationships between Terms

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pattern</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Noun₁+Noun₂</td>
<td>Parameter propertyOf Noun₁</td>
</tr>
<tr>
<td>2</td>
<td>Adjective+Noun</td>
<td>Parameter subClassOf Noun</td>
</tr>
<tr>
<td>3</td>
<td>Verb+Noun</td>
<td>Parameter subClassOf Noun</td>
</tr>
<tr>
<td>4</td>
<td>Noun₁+Noun₂+Noun₃</td>
<td>Parameter propertyOf Noun₁</td>
</tr>
<tr>
<td>5</td>
<td>Noun₁+Preposition+Noun₂</td>
<td>Parameter propertyOf Noun₂</td>
</tr>
</tbody>
</table>

Our ontology is generated from the set of parameters created in accordance with the above rules. Then, we are able to match a query and the ontology. 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 (i.e., Parameter propertyOf Noun), and (2) one concept is a subclass of the other concept (i.e., Parameter subClassOf Noun).

Using 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 relationship “X propertyOf Y” in its pattern rule, the matching logic will return a matching score because these two parameters are closely related.

3.3 Input/Output Semantic Matching

The semantic matching technique estimates the similarity of the input and output by considering the underlying concepts of the input/output parameters. Formally, we describe the input as a vector  \( I = \langle p_i, C_i \rangle \) (similarly, the output can be represented in the form  \( O = \langle p_o, C_o \rangle \)), where  \( p_i \) is the set of input parameters and  \( C_i \) is the set of concepts that are 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 divide  \( p_i \) into a set of terms and 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  \( A \) and  \( B \), the similarity between  \( A \) and  \( B \) is given by the following formula:

\[
Sim(A, B) = \frac{2 \times \|Match(A, B)\|}{n_1 + n_2}
\]

where  \( n_1 \) and  \( n_2 \) denote the number of valid terms in the parameters and  \( \|Match(A, B)\| \) returns the number of matching terms. The similarity of each parameter is calculated by the best matching parameter having a larger number of semantically related terms. The overall similarity is computed by a linear combination [6] to combine the similarity of each parameter.

Because matching techniques based on clustering consider all the 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 primary objective is to improve the precision of the matching technique by applying the pattern relationships defined in Table 2. For details, readers may refer to our previous work [6].

4. AUTOMATIC DISCOVERY AND COMPOSITION

Developing a data mashup is a process composed of the following phases: (1) suitable APIs must be selected from a registry, (2) IO mappings and pairing among different APIs must be established, and (3) programming code to actually combine the selected APIs must be written to complete the final application. This work focuses on phases (1) and (2).

4.1 Discovery Algorithm

Given a query and a collection of APIs stored in the registry, automatically finding an API from the registry that matches the query requirement is the Web API discovery problem. For example, assume that we are looking for an API to find a hotel. Fig. 1 shows the input/output parameters of a query and an API. In this example, a Web API \( W \) satisfies 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 a query parameter only if there is a semantic relationship between them. Here, although CountryCode and CountryID are different forms, they have the same semantics because they are referred to the same concept. In addition, NameOfCity and CityName have the same semantics because 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.

![Fig. 1. Example of Discovery Problem](image)

We describe an automatic Web API discovery algorithm similar to the one in [22]. An API matches a query when an API is sufficiently similar to the query. This means that we must allow the agent to perform matches that recognize the degree of similarity between the 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 requires to execute correctly. Our discovery algorithm is shown in Algorithm 1. This algorithm adopts strategies that rapidly prune the APIs that are guaranteed not to match the query, thus improving the efficiency of the system. A query is matched against all the 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 with the same process with the order of the query and API reversed. Consider a pair of candidate inputs/outputs: \( X = (x_1, x_2, \cdots, x_i, \cdots, x_m) \) and \( Y = (y_1, y_2, \cdots, y_i, \cdots, y_n) \), where \( x_i \) and \( y_j \) denote the parameters. The similarity 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

```
Result = ∅

For each W in registry
    If Matching(Q, W) then append W to Result
Endfor

Sort Result by SCORE
```

### Matching(Q, W) {
  Degree = SemanticMatch(Q.O, W.O)
  Degree = SemanticMatch(W.I, Q.I)
  SCORE = Degree + Degree
}

### SemanticMatch(X, Y) {
  Degree = 0
  If m=0 then return Degree=1
  For each \( x_i \) in X
    Find the BestValue(BV) of Sim(\( x_i, y_j \)) for all \( 1 \leq j \leq n 
    If BV=0 then return fail
    else Degree = Degree + BV
  Endfor
  Return Degree/m
}

#### 4.2 Composition Algorithm

Given a query and a collection of APIs, in the case that a matching API is not found, searching a sequence of APIs that can be combined is the composition problem of Web APIs. This 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. Fig. 2 shows the input/output parameters of a query \( Q \), and two Web APIs \( W_1 \) and \( W_2 \) in the registry. Assume that 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 figure, \( W_1 \)
returns HotelName as the output. \( W_2 \) receives this as the input and returns Location as the result. Therefore, the subsequent \( W_2 \) may use the output produced by the preceding \( W_1 \) as the input. We can define the Web API composition problem as Definition 3.

![Diagram](image)

**Fig. 2. Example of Composition Problem**

**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 its 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 finite sequence of APIs in the registry.

We describe a Web API as \( \langle W.I, W.O \rangle \) and a query as \( \langle Q.I, Q.O \rangle \). A composition is valid if the following conditions are satisfied: (1) \( \exists W_i(Q.I \supseteq W_i.I) \), (2) \( \exists W_j(Q.O \subseteq W_j.O) \), and (3) \( \forall W_i, W_j(W_i \rightarrow W_j) \): at least one path exists 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. The output from an API at any stage in the composition should be able to provide the input to the next API.

The composition problem is achieving a desired goal from the initial request without revealing the underlying composition details. Developers can now simply describe a goal in the form of a query and submit a requirement to our system. This composition consists of a set of workflow-like structures that control the execution of the Web APIs. Although many different control structures can be applied to the workflow, we take into account only two important structures: sequence and split. These structures allow us to build most of the possible compositions.

- **Sequence:** the output of an API is the input of one of following APIs. This is a basic control structure for the workflow.
- **Split:** two or more APIs are executed in parallel. As a result, it can produce several and different outputs.

**Construction of Composable Similarity Graph**

To accelerate the calculation of possible composition plans, we use a pre-computed CSG that chains the output of one API into the input of another API. This graph, \( G=(V, E) \), is based on a set of nodes, where \( V \) is the vertex set and \( E \) is the edge set. The con-
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. First, we assign each API $W$ in the registry to vertexes iteratively. We then establish edges between the vertexes. For each vertex $v_i$, we determine if its corresponding output can be accepted as an input by $v_j$ by computing the similarity score. If the output of $v_i$ is semantically similar to the input of $v_j$ (i.e., $\text{SemanticMatch}(v_j, i, v_i, O) > 0$), then we add a directed edge from $v_i$ to $v_j$ (in the reverse direction) and assign a similarity score. We also verify 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 CSG, we solve the composition problem within this graph. The initial graph is dynamically modified if new APIs become available.

**Algorithm 2: CSG Construction Algorithm**

```
For each W in registry
    $v_i = \text{addVertex}(W)$
Endfor
For each $v_i \in V$
    For each $v_j \in V$
        If $\text{SemanticMatch}(v_j, i, v_i, O) > 0$ then $\text{addEdge}(v_i, v_j)$
        If $\text{SemanticMatch}(v_i, i, v_j, O) > 0$ then $\text{addEdge}(v_j, v_i)$
    Endfor
Endfor
```

**Graph-based Composition Algorithm**

Our graph-based composition algorithm can be described as generating DAGs that can produce the output satisfying the desired goal. 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 BFS algorithm that is able to find the 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.

*Fig. 3. General Picture of Conceptual Approach*

*Searching sub-graphs:* First, we search the API registry for any API that has all the output parameters of the query (we call these “last nodes”), and any API that has at least
one of the input parameters of the query (we call these “first nodes”). Upon completion of this search, it is assumed that non empty sets are obtained for the first and last nodes. We then create \( n \)-ary trees for every node by visiting all the nodes connected to a particular last node. This tree is constructed by recursively including nodes and edges from the CSG 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. Fig. 3 shows a general picture of the conceptual approach we applied before constructing the overall composition plans.

**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 \( W_0 = <\emptyset, Q.I> \). \( W_0 \) is an API in a tree with no input, having only an output. Finding a possible composition candidate consists of 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 are validated in the next stage.

**Algorithm 3:** Graph-based Composition Algorithm

```plaintext
If SemanticMatch(Q.O, W.O) = 0 then fail
If SemanticMatch(W.I, Q.I) = 0 then fail
Searching Last Nodes and First Nodes
For each Last Node
    Call BFS algorithm
    Create \( n \)-ary trees
Endfor
For each \( n \)-ary 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 \( \land \) \( v_i = \emptyset \))
        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
```

**Validating candidates:** A possible composition candidate is valid if all the nodes in the composition can be executed non-sequentially in order to produce the desired results. Validating is performed by starting from the start node and working backwards. At this point, the first nodes consist of all the APIs such that all their inputs are provided by the start node. Let \( O_I \) be the union of all the outputs produced from the first nodes in the composition, and \( I_I \) (i.e., \( Q.I \)) be the query input. Inputs for the second nodes are all the outputs produced by the previous nodes and the query input, i.e., \( I_2 = O_I \cup I_I \). The com-
bination \( I_z \) will be the available input for the next nodes. This transition (i.e., \( I_{z+1} = O \cup I_z \)) is repeated until the last node is reached, removing redundant nodes that 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. That is, all the output parameters of the query must be obtained, and some or all of 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 to compute a similarity score. This score is calculated by the average value of all the similarity data related to the edges. 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 top of the list is considered the best, recommended option for the user. Algorithm 3 illustrates our graph-based composition algorithm.

### Implementation of Composition Engine

We have developed a semantic-based Web API composition engine. The system architecture is shown in Fig. 4. We select Java as the programming language because it allows for fast, flexible development. Our algorithms were implemented by using the publicly available *BreadthFirstPaths*, *CRFTagger*, *Apriori-T*, and *ClusterLib* tools. The composer is responsible for the plan to achieve the composition relevant to the desired goal. It captures the current composition states and dynamically composes the relevant APIs that should be added to the mashup. All composite APIs are manipulated in a DAG. In the DAG, each vertex corresponds to an API in the composition, and each edge corresponds to a part of the data-flow. Each incoming and outgoing edge represents the input and output of the API, respectively.

![Diagram of Web API Composition Engine](image)

**Fig. 4. Semantic-based Web API Composition Engine**

The discovery engine is used to select the individual APIs that are included in the composition. This engine is based on the semantic matching technique that allows users
to automate the discovery of Web APIs. In the graphical user interface (GUI), mashup developers can obtain the immediate composition results visually, and iteratively refine their goals until the results are satisfactory. The ontology learning method automatically builds semantic ontologies from the Web API descriptions. We consider the syntactic information that resides in the WSDL/WADL/XRDL file [6], and apply the hierarchical clustering and pattern analysis techniques to obtain their semantics.

We present a graph-based composition algorithm for the integration of the Web APIs in which a composition is gradually generated by forward and backward searching over the graph. At each step, suitable APIs are added to the composition. This process continues until the desired goal is satisfied. The identified APIs can be connected by data-flow arcs in a DAG automatically, thereby forming a data mashup without any major integration effort or the need for programming skills.

5. EXPERIMENTAL EVALUATION

To verify the effectiveness of our Web API composition engine we extracted a collection of REST and SOAP APIs from ProgrammableWeb. To avoid potential bias, we chose various APIs from different domains. We first collected a subset of REST APIs for three domains: weather, travel, and mapping. This set contained 56 APIs. Then, we collected a subset containing 17 SOAP APIs from three domains: zipcode, location, and search. In Fig. 5, we show the CSG obtained from our experimental dataset. The graph consists of 73 nodes and 149 edges.

Fig. 5. Composable Similarity Graph (CSG)
We chose a simple example of an automatic Web API composition to verify our solution. A possible query for the composition is given as follows: \( Q.I = \{ \text{zipcode} \} \) and \( Q.O = \{ \text{city, latitude, longitude} \} \). The composition result is represented by the part of the CSG shown in Fig. 6. As shown in Fig. 6(a), our engine has discovered five last nodes (three black circles and two yellow hexagons) and five first nodes (three grey circles and two yellow hexagons) from the registry. In this example, the yellow hexagons (node 07 and 70) satisfy the query requirement and the desired goal is directly matched to the output of a single API. Therefore, we move the two yellow hexagons to the result. For the remaining last nodes (i.e., node 11, 28 and 73), we call the BFS algorithm and create \( n \)-ary trees (see Fig. 6(b)).

Six possible composition candidates have been automatically generated from the graph (i.e., \( 72 \rightarrow 21 \rightarrow 11, 72 \rightarrow 67 \rightarrow 21 \rightarrow 11, 72 \rightarrow 67 \rightarrow 17 \rightarrow 73, 66 \rightarrow 21 \rightarrow 11, 66 \rightarrow 17 \rightarrow 73 \) and \( 44 \rightarrow 28 \)). As mentioned in Section 4.2, a start node \( W_0 \) is added to each tree and the validation of the candidates is performed to identify the optimal paths. At this point path \( 72 \rightarrow 67 \rightarrow 21 \rightarrow 11 \) is removed from the candidates. Upon completion of the validation, the final composition candidates are selected and similarity scores are calculated. In Table 3 we list these ranked composition candidates.

To evaluate the feasibility of composition whether or not the produced compositions meet our criteria, we verified the number of desired goals captured by the composition algorithm. It can be seen in Table 3 that four of the recommended results meet the desired or relevant goals. Although the fifth ranking result is determined to be invalid as it does not satisfy the user requirement, the top four ranking results satisfy the desired composition plans. Therefore its successful rate is close to 80%. These results show that our algorithm can generate most user-desired outputs.

### Table 3. List of Ranked Composition Candidates

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>DAG</th>
<th>Rank</th>
<th>Score</th>
<th>DAG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.72</td>
<td>( W_0 \rightarrow 44 \rightarrow 28 )</td>
<td>4</td>
<td>0.54</td>
<td>( W_0 \rightarrow 66 \rightarrow 21 \rightarrow 11 )</td>
</tr>
<tr>
<td>2</td>
<td>0.65</td>
<td>( W_0 \rightarrow 66 \rightarrow 17 \rightarrow 73 )</td>
<td>5</td>
<td>0.27</td>
<td>( W_0 \rightarrow 72 \rightarrow 67 \rightarrow 17 \rightarrow 73 )</td>
</tr>
<tr>
<td>3</td>
<td>0.58</td>
<td>( W_0 \rightarrow 72 \rightarrow 21 \rightarrow 11 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. CONCLUSIONS AND FUTURE WORK

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

Our future work will be focused on the investigation of other kinds of compositions with loops such as repeat-until and iterations. We are also exploring various optimization techniques that can be applied to the algorithm. For example, a heuristic AI planning technique could be used to find an optimized solution with a minimal number of paths. The use of dynamic optimization techniques with the graph could significantly improve the effectiveness and efficiency of our approach.

REFERENCES

1. http://www.programmableweb.com


