Semantic-Based Data Mashups Using Hierarchical Clustering and Pattern Analysis Methods*

YONG-JU LEE
School of Computer Information
Kyungpook National University
Sangju, 742-711 Korea

Data mashups enable users to create new applications by combining Web APIs from several data sources. However, the existing data mashup framework requires some programming knowledge, hence it is not suitable for use by non-expert users. In this paper, we present hierarchical clustering and pattern analysis methods that build semantic ontologies automatically, and propose similarity searching algorithms that support the operation semantic matching and composable API discovery. These algorithms allow mashup developers to automate the discovery and composition of Web APIs eliminating the need for programmer involvement. We describe an experimental study on a collection of 168 REST APIs and 50 SOAP APIs. The experimental results show that our approach performs better in terms of both the rate of recall and precision performance compared with existing methods.

Keywords: data mashups, Web API, hierarchical clustering, pattern analysis, similarity searching, semantic techniques

1. INTRODUCTION

A mashup is a Web application that integrates data, logic, and UI components from several different applications to create a new application. An example of a mashup is HousingMaps.com, 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, useful dataset. Data mashups have become very popular over the last few years; their use varies from addressing transient business needs in modern enterprises [1] to conducting scientific research in e-science communities [2]. In spite of their popularity, there are several challenging issues when combining Web APIs into data mashups, especially when the APIs use different protocols.

First, since a portal site may have a large number of APIs available for data mashups, manually searching and finding compatible APIs can be a tedious and time-consuming task. For example, as of August 2012, ProgrammableWeb.com has published more than 7000 APIs and it is possible to query this system to find available APIs. However, the portal site only supports keyword search and category search. Keyword searches are insufficient due to their bad recall and bad precision. Also, returned lists from category searches are generally based on criteria that have no relevance to mashup developers desired goals (e.g., in alphabetical or last update time order). Mashup develop-
users wish to quickly find the desired APIs and easily integrate them without having to expend considerable programming efforts.

Second, none of the existing portal sites provides a way to leverage semantic techniques that have been developed to assist users in locating and integrating APIs like those seen in traditional SOAP-based Web services. Unlike SOAP-based Web services, REST, JavaScript, XML-RPC, and Atom APIs do not use WSDL (Web Service Description Language) to specify their interfaces. Moreover, there is currently no established standard for categorizing and describing APIs for use in data mashups. Rather, APIs are discovered by developers who must search the Web manually. This is neither an easy nor manageable means of finding and composing APIs. Depending on the technology used, certain APIs may not be appropriate for use in a particular mashup application. These conditions lead to a tedious up-front process which could include repeated cycles of trial-and-error before the suitable APIs are found; this is especially true when the APIs are required to utilize a variety of protocols.

After suitable APIs have been discovered, the integration of these APIs should not require in-depth programming knowledge. Users should be able to integrate APIs with minimal training. A more challenging problem is to compose APIs dynamically, that is, on demand. The dynamic composition of APIs requires an understanding of the capabilities of those APIs (i.e., what they can do) and the compatibility between APIs.

In this paper, we explore the use of syntactic and semantic descriptions to find compatible APIs. We define API descriptions that syntactically describe Web APIs, and present hierarchical clustering and pattern analysis methods to semantically describe APIs. The hierarchical clustering method provides us with a breadth of coverage for common terms, whereas the pattern analysis method provides a depth of coverage by showing their relationships. By combining these two methods, we hope to improve both the recall and the precision performance of the search. We then propose a similarity searching method for Web APIs using these syntactic and semantic descriptions. Our approach suggests ranked solutions that meet the desired goals, and might provide more potentially relevant composition opportunities. It assists developers in iteratively refining their mashups for higher quality. The main contributions from this paper are as follows:

- This paper first shows that existing techniques and algorithms used for finding and matching SOAP-based Web services can be reused, with only minor changes, for the purpose of finding compatible APIs for data mashups. These APIs include REST, JavaScript, XML-RPC, and Atom APIs, which are not artifacts described in SOAP-based Web services.
- 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, and how to use the syntactic and semantic descriptions to aid the easy discovery and composition of Web APIs.

The rest of this paper is organized as follows. In section 2, we begin by investigating Web APIs. We describe the addition of semantics to Web APIs in section 3 and the similarity searching for Web APIs in section 4. We describe our experimental evaluation in section 5. Finally, we discuss related work in section 6, and conclude the work in section 7.
2. INVESTIGATING WEB APIS

Currently, the development of Web APIs is rather autonomous, guided by no established standards or rules, and Web API documentation is commonly not based on an interface description language such as WSDL, but is rather given directly in HTML as part of a Web page. As a result, the use of Web API requires extensive manual effort and the wealth of existing work on supporting common service task, including discovery, composition, and invocation, can almost never be reused or adapted to Web APIs.

Since ProgrammableWeb.com is currently the most popular API directory, we consider the following investigation to be representative of the current state of Web APIs. Fig. 1 shows the distribution of the different types of APIs in ProgrammableWeb.com. As can be seen, almost two third of the APIs (72%) are REST, 18% of the APIs are SOAP, 6% are JavaScript, 3% are XML-RPC, and 0% are Atom. Therefore, the Web APIs referred to in this paper contain REST, SOAP, JavaScript, and XML-RPC. In this section each API is examined in terms of its syntactic descriptions.

REST APIs are defined as services which conform to the REST (Representational State Transfer) paradigm [3]. REST is an architectural style for distributed hypermedia systems. REST APIs are commonly implemented by using HTTP, they comprise of a collection of uniquely identified resources and their links to each other. The resources in a REST API are both identified by and resolved using a URI (Uniform Resource Identifier). Resources are manipulated using a fixed set of four CRUD (Create, Read, Update, and Delete) operations: POST, GET, PUT, and DELETE. Representations of the resource show the current state of the resource’s data. One resource may have multiple representation formats such as HTML, XML, JSON, and so forth. REST APIs should also be stateless, meaning that no session state is stored on the server. Currently, we can use WADL [4] to syntactically describe REST APIs. WSDL 2.0 [5] also provides an HTTP binding extension to describe REST APIs.

In comparison to REST APIs, SOAP APIs do not directly use HTTP methods to access resources but rather define their own operations, wrapping the resource information, and then invoke these operations through a HTTP method. SOAP technology is well consolidated and has enabled the development of applications in many areas of business. Despite its relative success, SOAP has been heavily criticized because of its complexity and large number of related specifications. In essence, SOAP APIs expose their internal functionalities through a complex programming-language-like interface.
that is totally different to other APIs, while REST APIs expose internal data through a simple document-processing interface that is always the same. SOAP APIs are described by WSDL files, which contain information on how to access APIs and what operations are exposed.

To create a JavaScript API, a few additional steps are required beyond what is necessary to create a SOAP API. One of the first steps when implementing a JavaScript is to decide on the GUI technology, this may be based on HTML. The second step is to locate the data to be presented in the GUI. Often this information consists of records from a relational database. The third step is to write a piece of code to deal with the input and the output of the API. Up to this point, there is no difference in the process of creating a JavaScript or SOAP API. For the JavaScript API to be usable in a data mashup, the first additional step is to decide which inputs, outputs, and operations should be exposed. The second additional step is to generate a WSDL file that describes the exposed inputs, outputs, and operations. With the wide availability of Web service development toolkits, the generation of this WSDL file can be automated.

XML-RPC is a remote procedure call protocol that works over the Internet. An XML-RPC message is essentially a HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML. The XML-RPC API is simpler to use and understand than the SOAP API because: (1) It allows only one way of method serialization, whereas SOAP defines multiple different encodings; (2) it has a simpler security model; and (3) it does not require (or support) the creation of WSDL, although XRDL [6] provides a simple subset of the functionality provided by WSDL. However, the SOAP API is a successor of XML-RPC and as such it is more powerful. In this paper, we consider SOAP, JavaScript, and XML-RPC to be a family of RPC-style APIs.

In conclusion, REST, JavaScript, and XML-RPC APIs, like SOAP APIs, take programmatic inputs and produce programmatic outputs. Each type of Web APIs requires separate invocation support, which makes it even more challenging to provide support for the composition of APIs. Currently, mashup development is based on individual solutions, which have a low level of reusability and do not contribute to the automation of API composition.

3. ADDING SEMANTICS TO WEB APIS

Although programmatic inputs/outputs provide an easy and intuitive way 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. Another problem is that parameters only represent a flat structure with no hierarchy, thus the large number of parameters might cause difficulties for developers in matching compatible APIs.

3.1 Hierarchical Clustering Method

We present a hierarchical clustering method to derive several semantically meaningful concepts from parameters. We consider the syntactic information that resides in the WSDL/WADL/XRDL file, 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, and leverage these concepts to determine similarities between Web APIs. Formally, we can define an API as follows:

**Definition 1:** A Web API $W = \langle \exists, \forall \rangle$ where $\exists$ is a name and a text description that specify an API and $\forall$ is a set of operations provided by the API. An operation, $O = \langle x, \text{in}, \text{out} \rangle$, is described by a name and a text description $x$, and is associated with an input in and an output out. Each input and output contains a set of parameters.

From the above definition, we can identify three types of the API description. First, we note the name and text description which describes a general API. Second, we note the name and text description of the operation that captures the provided functionality. Finally, we note the parameters derived from the input/output. In this paper we focus on the operation matching problem, that is, given an API operation, how can we return a list of all similar operations.

To effectively locate operations of Web APIs, it is important to have their underlying semantics. However, this is difficult for two reasons. First, parameter naming is mostly dependent on the API developer’s habits. Hence, parameters tend to be subject to extreme variation, given the use of synonyms, hypernyms, abbreviations, etc. Second, input/output typically has only a few parameters, and the associated WSDL/WADL/XRDL files rarely provide rich descriptions for parameters. Traditional IR (Information Retrieval) techniques such as TF/IDF (Term Frequency/Inverse Document Frequency) [7] rely on word frequency to capture the underlying semantics, and thus are difficult to carry out.

As proposed by Dong et al. [8], we try to explore the underlying semantics of the inputs/outputs in addition to the textual descriptions. The input/output parameters are often combined as a sequence of several terms. For example, take a parameter "ArrivalTimeOfAirplane," the terms are specified by their first letter capitalized \{Arrival, Time, Of, Airplane\}. We cluster these terms into several concepts. In our opinion, considering the terms with the concepts they may be able to significantly improve the quality of operation searching. For example, given the two outputs \{Weather\} and \{Temperature, Humidity\}, they cannot be considered to be similar just by referencing their names. But these terms are all related to the concept of “Weather,” so they should be similar parameters.

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 [8]. This allows us to cluster terms by exploiting the conditional probability of their occurrence in the inputs and outputs of APIs.

Let $T = \{t_1, t_2, ..., t_m\}$ be a set of terms. Let $IO$ be a set of candidate inputs/outputs available from API descriptions. To reflect co-occurrence, we introduce association rules [9] 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 $s$ and confidence $c$.

**Definition 2:** The support $s(t_i) = P(t_i) = ||IO(t_i)||/||IO||$ is the probability that $t_i$ occurs in an input/output, where $||IO||$ is the total number of inputs and outputs of APIs, and $||IO(t_i)||$ is the number of inputs and outputs that contain $t_i$. 

---

**Footnote:**

[7] Term Frequency/Inverse Document Frequency

[8] Dong et al.

[9] Association rules
Definition 3: The confidence $c(t_i \rightarrow t_j) = P(t_j|t_i) = \frac{||IO(t_i \cup t_j)||}{||IO(t_i)||}$ is the probability that $t_j$ occurs in an input/output, given $t_i$ is known to occur in it, where $||IO(t_i \cup t_j)||$ is the number of inputs and outputs that contain both $t_i$ and $t_j$.

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., $c(t_i \rightarrow t_j) > \delta$). Ideally, parameter clustering results should have the following two features: The cohesion within a concept (connections between parameters inside the concept) should be strong, and the correlation between concepts (connections between parameters in different concepts) should be weak.

Definition 4: Given a cluster $C_1$, we define the cohesion of $C_1$ as the percentage of term pairs closely associated over all term pairs:

$$\text{cohesion}(C_1) = \frac{||\{t_i \rightarrow t_j | c(t_i \rightarrow t_j) > \delta\}||}{||C_1||}$$

where $t_i, t_j \in C_1, t_i \neq t_j$.

Definition 5: Given clusters $C_1$ and $C_2$, we define the correlation between $C_1$ and $C_2$ as the percentage of closely associated cross-cluster term pairs:

$$\text{correlation}(C_1, C_2) = \frac{||\{t_i, t_j | c(t_i \rightarrow t_j) > \delta\}||}{2||C_1||}$$

where $t_i \in C_1$, and $t_j \in C_2$.

To measure the overall quality of a clustering, we define score $= \frac{\text{coh}}{\text{cor}}$ 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.

3.1.1 Clustering algorithm

The traditional clustering algorithm does not perform well when used with our method. In general, the cohesion is defined as the sum of squares of Euclidean distances from each point to the center of the cluster it belongs to; the correlation is defined as the sum of squares of distances between cluster centers. This definition does not perform well in our method because of “the curse of dimensionality.” WSDL/WADL/XRDL files in a repository may contain thousands of terms as features or dimensions. As the number of dimensions increases, the data become increasingly sparse, such that the distance measurements between the pairs of points become meaningless and the average density of the points anywhere in the data is likely to be low. Therefore, a different clustering methodology must be developed for high-dimensional data.

The frequent pattern-based clustering technique extracts distinct frequent patterns among subsets of dimensions which frequently occur. It employs such patterns to group objects and generate meaningful clusters. We quantify the cohesion and correlation of clusters based on the frequent termsets. Each input/output can be represented as a set of terms. Collectively, a large set of inputs/outputs will contain a very large set of distinct terms. If we treat each term as a dimension, the dimension space will evidence very high
dimensionality. This difficulty can be overcome by association rules. The problem of mining association rules can be regarded as a two-step process: (1) Find all frequent termsets (e.g., {city}, {country, city}, {zip, area, state}, etc.); and (2) generate strong association rules (e.g., zip \(\rightarrow\) code, s: 0.3, c: 0.9) from the frequent termsets. This problem can be computed by using the well-known Apriori algorithm [10]. That is, we can mine a set of frequent termsets from the set of inputs/outputs. Infrequent rules with less than a minimum support and minimum confidence are discarded.

We use a refinement of a classical agglomerative hierarchical clustering algorithm [11] to turn the set of terms \(T = \{t_1, t_2, \ldots, t_m\}\) into the concepts \(C = \{c_1, c_2, \ldots, c_n\}\). The agglomerative hierarchical clustering is formed via a bottom-up fashion. This bottom-up strategy begins by placing each object in its own cluster and merges these atomic clusters into larger and larger clusters until no more clusters can be merged. In the context of this study, we sort the association rules in descending order first by confidence and then by support. At each step, the algorithm selects the highest ranked rule that has not been previously considered. If the terms in the rule belong to different clusters, the algorithm merges the clusters (e.g., \{zip, code\}). We then compute the coh/cor score. There are two options; one is the score for the merged result in the previous step, and the other is the score for a new result reflecting the merger of two similar clusters. We select the option with a higher score. This process is repeated until eventually all result clusters satisfy the best score. For example, our algorithm can generate final clusters such as \{zip, code\}, \{country, city\}, \{zip, city, area, state\}. Fig. 2 illustrates the hierarchical clustering algorithm.

**Algorithm 1:** Hierarchical Clustering

1. Find all frequent termsets
2. Generate association rules \(R\)
3. Sort \(R\) in descending order first by confidence and then by support
4. For each association rule in \(R\)
   - Choose the highest ranked rule that has not been considered
   - If terms in the rule belong to different clusters
     - then merge the clusters
5. Repeat
6. Compute coh/cor score (there are two options)
7. Choose the option with a higher score
8. Until all result clusters satisfy the best score

---

**3.2 Pattern Analysis Method**

Our pattern analysis method captures the 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 [12]. In order to characterize these patterns, 8209 parameters from 168 REST APIs and 1245 parameters from 50
SOAP APIs pulled off the Internet were categorized into several buckets. As shown in Table 1, 2435 (30%), 752 (9%), 608 (7%), 472 (6%), and 368 (5%) parameters in 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 3574 (43%) parameters not covered by any of the patterns. The majority of them (3548 parameters) contain only one token (e.g., City), the others (26 parameters) cannot be tokenized according to the patterns defined in Table 1. This table also shows the pattern analysis results with respect to SOAP APIs. 470 (37%), 175 (14%), 70 (6%), 40 (3%), and 15 (1%) parameters in 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. Consequently, we can learn that there are five common patterns for REST and SOAP APIs, although the rate of their occurrence is different depending on the protocol.

Table 1. Pattern analysis for REST and SOAP APIs.

<table>
<thead>
<tr>
<th>No</th>
<th>REST APIs</th>
<th>SOAP APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Pattern</td>
<td>Occurrence</td>
</tr>
<tr>
<td>1</td>
<td>Noun$_1$+Noun$_2$</td>
<td>2435 (30%)</td>
</tr>
<tr>
<td>2</td>
<td>Adjective+Noun</td>
<td>752 (9%)</td>
</tr>
<tr>
<td>3</td>
<td>Verb+Noun</td>
<td>608 (7%)</td>
</tr>
<tr>
<td>4</td>
<td>Noun$_1$+Noun$_2$+Noun$_3$</td>
<td>472 (6%)</td>
</tr>
<tr>
<td>5</td>
<td>Noun$_1$+Preposition+Noun$_2$</td>
<td>368 (5%)</td>
</tr>
</tbody>
</table>

The first step in this method is to capture the relationships between the terms in a parameter and save them in the ontology. From the above table, the patterns can be classified into one of the two following categories:

Category 1: Noun$_1$+(Preposition)+Noun$_2$+ ... +Noun$_n$

Category 2: Adjective/Verb+Noun

We discover dependency relationships of a given parameter. A dependency relationship is an asymmetric binary relation between a word called head and the remaining multiple-words named modifier, where head reveals the most emphasized word of the phrase [13] (e.g., HighTemperature would be resolved to pair {head: Temperature, modifier: High}). We rely on the following rules to exploit output of dependency parsing of each term to capture ontological classes and object property relationships. The construction of these rules is based on the following observations:

- Single-word terms denote broader concepts than multiple-words phrases. Let’s consider two concepts X and Y. X is a head word and Y is a newly created phrase by adding a modifier segment to X. In this case, X is the hyper concept of Y (corresponding in the literature to the well-known subclass relationship).
- The relationship between head and modifier segment in case of Category 1 resembles, from grammatical point of view, a kind of possessive authority for the head segment over an entire term. Based on this observation, the property relationship between the two discovered concepts is asserted.
Based on the aforementioned observations, our pattern analysis rules are defined in Table 2.

Our ontology is generated from the set of parameters created in accordance with the above rules. The next step is to match a query and the ontology.

**Definition 6:** 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).

Based on the above rules, a search engine would be able to find a match based on the similarity of the operation. 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 because the two parameters are not equal (*i.e.*, “CityName” != “CodeOfCity”). 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”).

### 4. SIMILARITY SEARCHING FOR WEB APIs

In this section, we describe how to predict the similarity of Web APIs. The addition of semantics to Web APIs described in the previous section has a unique advantage in that any additional semantics added to the syntactic information do not change the structure of the API description. Therefore, existing semantic Web service searching algorithms can be used directly for matching APIs. We explore the use of syntactic and semantic descriptions to find matching APIs. Syntactic descriptions are derived using a thesaurus (in this case WordNet [14]) after tokenization, POS (Part-Of-Speech), stopword filtering, and abbreviation expansion. Semantic descriptions are derived by the hierarchical clustering and pattern analysis methods. Matches, due to the two cues, are combined to determine an overall similarity score.

<table>
<thead>
<tr>
<th>No</th>
<th>Pattern</th>
<th>Example</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Noun1+Noun2</td>
<td>CountryID</td>
<td>Parameter <code>propertyOf</code> Noun1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(CountryID <code>propertyOf</code> Country)</td>
</tr>
<tr>
<td>2</td>
<td>Adjective+Noun</td>
<td>HighTemperature</td>
<td>Parameter <code>subClassOf</code> Noun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(HighTemperature <code>subClassOf</code> Temperature)</td>
</tr>
<tr>
<td>3</td>
<td>Verb+Noun</td>
<td>EnrollAccount</td>
<td>Parameter <code>subClassOf</code> Noun</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(EnrollAccount <code>subClassOf</code> Account)</td>
</tr>
<tr>
<td>4</td>
<td>Noun1+Noun2+Noun3</td>
<td>TelephoneAreaCode</td>
<td>Parameter <code>propertyOf</code> Noun1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(TelephoneAreaCode <code>propertyOf</code> Telephone)</td>
</tr>
<tr>
<td>5</td>
<td>Noun1+Preposition+Noun2</td>
<td>TimeOfDeparture</td>
<td>Parameter <code>propertyOf</code> Noun2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(TimeOfDeparture <code>propertyOf</code> Departure)</td>
</tr>
</tbody>
</table>
4.1 Semantic Matching Algorithm

We use a semantic matching algorithm similar to the one in [15]. The input of this matching algorithm is a single query and a collection of API descriptions. The output is a list of target operations sorted by the ratio of matched tokens. In order to calculate a matching score, both the query and the API descriptions must be preprocessed using the following procedure:

- **Tokenization:** After parsing the API descriptions, if a parameter contains multiple-terms, the parameter is tokenized. We find word boundaries within the parameter using changes in font and the presence of delimiters such as underscore, hyphen, and numeric to alphanumeric transitions. Thus a parameter such as “ClientName” will be separated into “Client” and “Name.”

- **POS and stop-word filtering:** Simple grammar rules are used to detect noun phrases, adjectives, and verbs. Stop-word filtering is performed using pre-supplied list. We have used stop words similar to those used in general search engines, including words such as “and, or, the, is,” etc.

- **Abbreviation expansion:** A list of abbreviations and their expansions is used to expand tokenized words that are abbreviated. For example, “ID” is expanded into “Identification.”

- **Synonym search:** A thesaurus (i.e., WordNet) is employed to find synonyms. For example, the term “Town” is included as a synonym for the tokenized word “City.”

In the hierarchical clustering method, we clustered concepts as a baseline to measure the similarity of operations. As defined in Definition 1, an operation $O$ is a vector $O = \langle x, in, out \rangle$. Given two operations $(A, B)$, we can determine the similarity by combining the similarities of individual vector elements.

First, we estimate the similarity of the text description, it can be achieved by employing the traditional TF/IDF method. Next, we estimate 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 $in = (p_{in}, C_{in})$ (similarly, the output can be represented in the form of $out = (p_{out}, C_{out})$), where $p_{in}$ is the set of input parameters and $C_{in}$ is the set of concepts that are associated with $p_{in}$. 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 $p_{in}$ into a set of terms, then we 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 inputs. Consider a pair of candidate inputs: $A_{in} = (a_1, \ldots, a_i, \ldots, a_m)$ and $B_{in} = (b_1, \ldots, b_j, \ldots, b_n)$, where $a_i$ and $b_j$ denote the parameters. The similarity between $A_{in}$ and $B_{in}$ is given by the following formula:

$$Sim(A_{in}, B_{in}) = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} \text{Match}(a_i, b_j)}{m + n}$$

where $\text{Match}(a_i, b_j) = \max\{\text{TF/IDF}(a_i, b_j)\}$ for all $1 \leq i \leq m$, $1 \leq j \leq n$.

We use a linear combination to combine the similarity of each component. Each
similarity is assigned a weight which is based on the user’s confidence. The overall similarity is calculated by using \( Sim_x \), \( Sim_{in} \), and \( Sim_{out} \):

\[
Similarity(A, B) = w_1(Sim_x) + w_2(Sim_{in}) + w_3(Sim_{out})
\]

where \( Sim_x \), \( Sim_{in} \), and \( Sim_{out} \) are the description, input, and output similarity, respectively. These functions return a real value between 0 and 1, indicating the degree of similarity. \( w_k \) (\( k = 1, 2, 3 \)) is the weight assigned to each similarity of description, input, and output; it indicates the degree of confidence. High weight values indicate the user’s confidence. For example, let us consider a description and an input of the operation. If the user is not confident about the description given, the weight can be set to a low value (e.g., 0.2). If the user is certain of the input given, the value can be set to 0.8. In our experiments we use equal weights; please note \( \sum w_k \leq 1 \).

Since the semantic matching algorithm based on clustering considers all terms in a cluster as an equivalent concept and ignores 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 semantic matching algorithm by applying the pattern rules defined in Table 2. If two concepts are matched according to the rules, the weighting is set to 1. If two concepts are not matched, then the weighting is set to 0 and remove them from the result. The core procedure for the semantic matching algorithm is shown in Fig. 3. We describe an operation \( O \) as \( \langle O.x, O.in, O.out \rangle \) and a query \( Q \) as \( \langle Q.x, Q.in, Q.out \rangle \). In order to speed up the calculation of matching operations, we use a repository that contains pre-computed concepts and pattern rules for a large collection of available operations.

**Algorithm 2: Semantic Matching**

A query \( Q \) is preprocessed using thesaurus, POS, stop-word and abbreviation lists

For each operation \( O \) in the repository

- Estimate \( Sim_x = \frac{TF}{IDF}(Q.x, O.x) \)

For each term in the query input/output

- clusterTerms = FindClusterTerms(term), replace term with clusterTerms

Endfor

- Compute \( Sim_{in} = Sim(Q.x, O.x) \) and \( Sim_{out} = Sim(Q.out, O.out) \), including the pruning process

Endfor

Overall similarity \( Similarity(Q, O) \) is calculated using \( Sim_x, Sim_{in} \) and \( Sim_{out} \)

Sort result operation list in descending order by the degree of similarity

**FindClusterTerms**

\( \{ \)

- clusterTerms = \( \emptyset \)

For each \( c_i \) in concepts \( C \)

- If \( c_i \) includes term then all terms in \( c_i \) are added to clusterTerms

Endfor

- Return clusterTerms

\( \} \)

Fig. 3. Semantic matching algorithm.
4.2 Composable API Discovery Algorithm

We have implemented an algorithm to support the integration of APIs using semantic descriptions. The filtering and selection of APIs is achieved by using the composable API discovery algorithm, which is similar to one implemented in the OWL-S Matchmaker [16]. Firstly, we can define the matching criteria as follows:

Definition 7: An operation matches a query when all the output parameters of the query are matched by the output parameters of the operation, and all the input parameters of the operation are matched by the input parameters of the query.

Definition 7 guarantees that the matched operation satisfies the needs of the query, and that the query provides to the matched operation all the input parameters that it needs to operate correctly [16]. The main procedure of the composable API discovery algorithm is shown in Fig. 4. A query is matched against all operations stored in the repository. Whenever a match between the query and any of operations is found, it is recorded and sorted within the matches according to highest score. A match between a query and an operation consists of matching the output of the query against the output of the operation, and the input of the operation against the input of the query. If a parameter of the query's output is not matched by any parameter of the operation's output, the match fails. Matching between inputs is computed by the same process, but with the order of the query and the operation reversed. The degree of a match between two parameters is calculated by the $\text{Match}(a_i, b_j)$ described in section 4.1. $A.io$ and $B.io$ here mean $A_{in} = (a_1, a_2, ..., a_i, ..., a_m)$ and $B_{in} = (b_1, b_2, ..., b_i, ..., b_n)$ (or $A_{out}$ and $B_{out}$).

![Algorithm 3: Composable API Discovery](image)

5. EXPERIMENTAL EVALUATION

We set up a series of experiments to evaluate our approach. The experiments contain comparing recall/precision of our approach with that of some existing work. Our objective is to show that we can achieve high recall and high precision performance in
finding similar operations, and to investigate the contribution of the two different methods (i.e., hierarchical clustering and pattern analysis) of our approach.

5.1 Dataset Preparation

To run our experiments, we extracted a collection of REST and SOAP APIs from the ProgrammableWeb.com and Xmethods.net, respectively. To avoid potential bias, we chose different APIs from different domain categories. We first collected a subset which associated REST APIs for three domains: mapping, travel, and weather. The ProgrammableWeb.com allows us to manually extract information regarding their operation descriptions, inputs, and outputs. This set contains 168 APIs and 264 inputs/outputs. There are a total of 501 terms. During our experiments, while manually evaluating the clustering results, we observed that a minimum support of 3% and a minimum confidence of 80% are reasonable threshold values. Next, we collected a subset containing 50 SOAP APIs from three different domains: zipcode, weather, and address. There are a total of 85 inputs/outputs with 104 terms. Appropriate threshold values are found for a minimum support of 10% and a minimum confidence of 80%.

We measured overall performance using recall and precision. Recall is a measure of completeness, whereas precision is a measure of exactness or fidelity. An inverse relationship exists between recall and precision, in which it is possible to increase one only at the cost of reducing the other.

Definition 8: Recall \( R = \frac{A}{A + B} \), where \( A \) stands for the number of returned relevant operations, \( B \) stands for the number of missing relevant operations, and \( A + B \) stands for the total number of relevant operations.

Definition 9: Precision \( P = \frac{A}{A + C} \), where \( A \) stands for the number of returned relevant operations, \( C \) stands for the number of returned irrelevant operations, and \( A + C \) stands for the total number of returned operations.

5.2 Results and Analysis

Our experiments compared our hierarchical clustering method with the pattern analysis method, which we refer to as **SEMANTIC**, with **KEYWORD** and **CLUSTER**. **KEYWORD** and **CLUSTER** denote the traditional keyword searching method and the hierarchical clustering method without the pattern analysis method, respectively. The reason for testing our approach with or without the pattern analysis method is to assess the impact of the ontological pruning process. We plot the recall-precision curve (R-P curve). In an R-P curve, the X-axis represents recall, and Y-axis represents precision. The curve closest to the upper right-hand corner of the graph indicates the best performance. The R-P curve is regarded by the IR community as the most informative graph demonstrating the efficacy of a search engine. We plotted the average values for different twenty queries in Fig. 5. From this figure, we can observe that **SEMANTIC** and **CLUSTER** methods exhibit better performance than the **KEYWORD** method. The reason is that both try to attach the underlying semantic concepts to the input/output. In particular, our approach (i.e., **SEMANTIC**) exhibits the best performance with its pattern analysis method, for the
ontological pruning makes it possible to search only relevant operations to ensure high performance.

We present recall and precision values of top-3, top-5, and top-10 returned results found via each method. Top-\(k\) means the \(k\) ranked similar operations; 1st ranking operation is considered the best recommended option for the user. Fig. 6 illustrates top-\(k\) recall comparison. To measure recall, we are interested in the number of returned relevant operations that can be returned within a given set of relevant operations. We can see that both SEMANTIC and CLUSTER can achieve satisfying results. For top-3, top-5, and top-10 recalls, SEMANTIC reaches 90\%, 79\%, and 69\%, respectively, obviously higher than the other two methods. CLUSTER is a little lower at the level of top-3, top-5, and top-10, as it measures the similarity based on the underlying semantics. KEYWORD has very low recall, because it only matches the keywords from the user’s requests, and the results are coarse.

Fig. 7 presents top-\(k\) precision comparison. As shown in this figure, SEMANTIC beats the other two methods in each result. For top-3, top-5, and top-10 returned results, SEMANTIC precision is 90\%, 82\%, and 69\%, respectively. SEMANTIC can improve precision 22\%, 27\%, and 31\%, respectively, rather than those of KEYWORD, 21\%, 25\%, and
27%, respectively, higher than those of CLUSTER. The figure demonstrates that with the expected number of returned results increasing, the rate of performance improvement increases. CLUSTER significantly improves recall performance, but only improves precision performance slightly. One major cause of this is the clustering terms exert two-fold effects: On the one hand, they provide additional evidence, which significantly assists in the number of returned relevant operations. On the other hand, they harbor noise that dilutes the high-quality evidence, whereas SEMANTIC rapidly filters out irrelevant concepts. In conclusion, SEMANTIC outperforms KEYWORD and CLUSTER, since it can improve both recall and precision performance significantly.

6. RELATED WORK

Mashups have received wide research interest in the last few years. A number of mashup approaches, techniques, and tools have emerged both for commercial and personal purposes. For example, Damia, Mashup Hub, Kapow Mashup Server, and Presto Edge largely target enterprise intranet environments, whereas Pipes, Popfly, and MashMaker are aimed at individual users and private use [17]. From a data mashup perspective, although research has been suggested to aid rapid mashup development, most work provides limited search and integration capabilities for mashups and users are still required to know how to write code and link the APIs using technical concepts derived from programming.

Yahoo Pipes [18] allows data mashups that combine RSS, Atom, or RDF formatted feeds. It provides a visual tool for data extraction and processing of Web sources. Pipes focuses on merging and transforming feeds via Web service calls. IBM’s Damia [19] extends the type of data sources for mashups to enterprise types such as Excel, Notes, XML, and relational databases rather than just URL-based sources as in Yahoo Pipes. Intel’s MashMaker [20] observes the user's behavior (e.g., what kind of data s/he is interested in) and recommends an existing mashup that the user would find useful. It also correlates the user's behavior with that of other users and uses the knowledge to suggest mashups defined by other users on the same web page.

For semantic-based data mashups, Semantic Web Pipes [21], which is inspired by Yahoo Pipes, offers a tool to build RDF-based mashups and specialized operators to cre-
ate semantic Web applications. MARIO [22] proposes a tag-based navigation technique to compose data mashups. Its engine allows users to explore the space of potentially composable data mashups and preview composition results as they interactively refine their “wishes,” i.e., mashup composition goals. MARIO offers a lightweight planner that enables the search abstraction for automatic flow composition. MatchUp [23] introduces an autocompletion mechanism. The key observation guiding the development of MatchUp is that mashups developed by different users typically share common characteristics; they use similar classes of mashup components and glue them together in a similar manner. These approaches share similar goals as ours in providing high-level semantic matching techniques for users to choose suitable components for a given situation.

Smashups [24] uses RESTful semantic Web services that are semantically annotated using SA-REST. The role of SA-REST is to enable automatic data mediation. Semantic annotations provide an agent with the ability to know more about an API’s input/output and what the API does, such that the agent can automatically compose Web APIs without human intervention. iMashup [25] proposes an approach for composing data-driven mashups, based on tag-based semantics. Lotus Expeditor Composer [26] enables progressive composition of non-Web-service-based components such as portlets, Web applications, native widgets, legacy systems, and Java Beans. These works are similar to our study. However, they do not support the automatic construction of semantic ontologies.

7. CONCLUSIONS AND FUTURE WORK

This paper presents hierarchical clustering and pattern analysis methods to semantically describe Web APIs. The hierarchical clustering method groups parameters in the collection of APIs into semantically meaningful concepts. The pattern analysis method captures the relationships between the terms contained in a parameter. We propose a set of similarity searching algorithms to improve the recall and precision performance of Web API search. The proposed algorithms provide the means for operation semantic matching and composable API discovery. These are both crucial features in data mashup research. The experimental results show that our approach significantly improves recall and precision performance compared with the traditional keyword searching method. The results demonstrate that our approach achieves up to 29% improvement for recall performance, and up to 31% for precision performance relative to the keyword searching method.

We are currently developing a mashup system for the automated APIs composition [27]. More specifically, we are working on a goal-directed interactive composition system for the integration of Web APIs, where the final mashup is gradually generated by a forward-backward chaining of APIs. At each step, a new API is added to the composition. However, current API compositions in our system require a human who has the domain knowledge and can guide the overall composition process. The incorporation of Artificial Intelligence based planning technology would result in further automation of the system. The ability to access machine understandable data via semantic Web is expected to make it easier to integrate a planner into this system.
REFERENCES

Yong-Ju Lee received the Ph.D. degree in Information and Communication Engineering, KAIST (Korea Advanced Institute of Science and Technology) in 1997, South Korea. He is now a Professor in School of Computer Information, Kyungpook National University, Sangju, Korea. He has published more than 60 papers in domestic and international conferences and journals. His current research interests include web services, cloud computing, semantic web technology, and mobile applications.