Extracting Business Execution Processes of API Services for Mashup Creation

Guobing Zou[1,2], Yang Xiang[1,2], Pengwei Wang[3], Shengye Pang[1], Honghao Gao[1], Sen Niu[4(✉)], and Yanglan Gan[3(✉)]

[1] School of Computer Engineering and Science, Shanghai University, China
[2] Shanghai Institute for Advanced Communication and Data Science, Shanghai University, China
{guobingzou,yangxiang618}@gmail.com
[3] School of Computer Science and Technology, Donghua University, China
{wangpengwei,ylgan}@dhu.edu.cn
[4] School of Computer and Information Engineering, Shanghai Polytechnic University, China
sens306314@gmail.com

Abstract. Mashup services creation has become a new research issue for service-oriented complex application systems. During the mashup service creation, how to extract business execution processes among APIs plays an important role when a mashup service developer receives a bunch of recommended API services. However, it does not exist an effective way to perform mashup recommendation with the support of extracting API business execution processes. In this paper, we propose a novel approach for automated extraction of API business execution processes for mashup creation. Based on the proposed word-domain matrix model, API annotation in a mashup service is transformed as a bipartite graph problem that is solved by the maximum bipartite matching algorithm to semantically annotate involved APIs. Then, directed dependency network among APIs is constructed by analyzing path dependencies and evaluating the compound polarity. Finally, API business execution processes in a mashup service can be extracted. The advantage of the work is that it generates business execution processes instead of a list of independent APIs, which can significantly facilitate mashup service creation for software developers. To validate the performance, we conduct extensive experiments on a large-scale real-world dataset crawled from ProgrammableWeb. The experimental results demonstrate the feasibility and effectiveness of our proposed approach.

Keywords: Service-oriented computing· API service· Mashup creation· Business execution processes· API annotation.

1 Introduction

With the advancement of network technology and increasing demands on service-oriented application integration, more and more service providers publish their software on the Internet in the form of web APIs. It accelerates the interoperable

machine-to-machine interaction and greatly promotes the procedure of service discovery, optimum selection, automatic composition and recommendation. As of May 2018, the world's largest online service repository ProgrammableWeb recorded more than 19,000 API services and approximately 8,000 mashup services. Especially, developing a mashup service for software engineers who use multiple individual existing APIs as components to create a value-added composite service becomes a popular software development schema in service-oriented environment [12]. Mashup services integrate the data and functionality from more than one APIs that enriches the applicability of web services. Currently, most existing mashup services are created by software developers who manually choose appropriate APIs and compose them together as a whole to a service management platform. As a result, it tends to be a labor-intensive challenging task for mashup service developers to select their desired web APIs from multiple functionally equivalent candidate ones in a large service repository.

To address the provision of web APIs, correlative research efforts have been made to improve the effectiveness of mashup creation. The mainstream method includes semantic-based, social network-based and machine learning-based service recommendation. The semantic-based methods [5,6,8,12] mainly focus on using LDA probabilistic topic model to calculate semantic similarity between mashup request and API description. Another way [1,11,12] is to leverage social network techniques to mine user's social features and interests from usage historical logging, where candidate APIs are recommended according to their high similarity of the social aspects with users. Moreover, machine learning algorithms [9,10,13] such as clustering and matrix factorization are recently exploited to more effectively enhance API recommendation.

Although the above investigation can assist and facilitate the procedure of recommending appropriate APIs from a large-scale web service repository, the deficiency of the existing approaches is that they are still difficult and time-consuming for software developers to create a mashup service. The reason is that they need to further understand the functionality of the provided web APIs and their corresponding business invocation relationships, when these APIs are programmed and integrated into a mashup service. Therefore, how to design a novel approach for automatically and effective extraction of business execution processes among APIs has been a key research issue to be solved in mashup service creation.

An idea way of overcoming the above problem is to reason out the business execution processes for a set of APIs that are recommended to mashup developers. To this end, we propose a novel framework for automated extraction of API business execution processes when developing a mashup service. Given a mashup service repository, a word-domain matrix is firstly modeled through calculating the semantic similarity between word and domain by WordNet. In such case, we then transform an API annotation problem to a weighted bipartite graph, where the maximum bipartite matching algorithm is employed to optimally find a solution to semantically annotate APIs in a mashup functional description. Afterwards, a directed dependency network is constructed among APIs by analyzing path dependencies and evaluating the compound polarity via Stanford

CoreNLP parser. Finally, business execution processes among APIs can be extracted by network maximum flow algorithm. The advantage of the work is that it generates API business execution processes based on an existing service recommendation approach that only produces a list of independent APIs. Therefore, it can significantly facilitate mashup service creation for software developers. To validate the feasibility and effectiveness of our approach, we conduct extensive experiments on a real-world dataset crawled from ProgrammableWeb. The experimental results demonstrate that our approach outperforms the competing ones in terms of six evaluation metrics.

The main contributions of this paper are summarized as follows:

- We propose a novel API annotation approach that semantically maps highly correlative words to their corresponding APIs in mashup functional description, where word-domain matrix is constructed to provide weights in the modeled bipartite graph.
- On the basis of API annotation, we propose a novel approach for extracting API dependency network in mashup service by analyzing path dependencies between APIs and evaluating their compound polarity.
- We design and implement a prototype system and conduct extensive experiments on a large-scale real-world dataset crawled from ProgrammableWeb. The experimental results validate the feasibility and effectiveness of our proposed approach for business execution processes extraction.

The reminder of this paper is organized as follows. The problem is formulated in Section 2. Section 3 elaborates our approach for extracting API business execution processes. Section 4 presents extensive experiments and analyzes the performance. Section 5 reviews the related work. Finally, Section 6 concludes the paper.

2 Problem Formulation

Definition 1 (API Service). A web API can be denoted as a two-tuple $api = \langle W^{(a)}, D^{(a)} \rangle$, where $W^{(a)} = \{w_1, w_2, \cdots\}$ is a functional description, and $w_i (i = 1, 2, \cdots)$ is the i-th word in the description. $D^{(a)} = domain_a$ corresponds to a domain tagged in api.

Definition 2 (Mashup Service). A mashup service, M, is represented as a three-tuple $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, where $W^{(m)} = \{w_1, w_2, \cdots\}$ is a functional description. $L^{(m)} = \{api_1, api_2, \cdots\}$ is a list of APIs, where api_i is the i-th API involved in M. $D^{(m)} = domain_m$ corresponds to a domain tagged in M.

Definition 3 (Atomic Grammatical Dependency). Given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, the atomic grammatical dependency of any two words w_s and w_t in $W^{(m)}$ is reflected by a set of directed paths denoted as $dep(w_s, w_t)$

$$dep(w_s, w_t) = \{w_s \xrightarrow{td_{z1}} w_{z1} \xleftarrow{td_{z2}} w_{z2} \xrightarrow{td_{z3}} \ldots \xrightarrow{td_{zn}} w_t\} \quad (1)$$

Where w_{zi} is the i-th dependency bridge word. td_{zj} represents a binary dependency relationship of the two adjacent words.

Note that the arrow direction of the dependency relationship indicates the order of dependency or domination between two adjacent words. The forward arrow $\mathbf{w_s} \rightarrow \mathbf{w_t}$ states the dependency of w_s on w_t, or w_t is dominated by w_s. Conversely, the reverse arrow $\mathbf{w_s} \leftarrow \mathbf{w_t}$ states the dominance of w_s on w_t, or w_t depends on w_s.

Definition 4 (Grammatical Dependency Set). Given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, the set of grammatical dependencies $C^{(m)}$ in $W^{(m)}$ can be expressed as

$$C^{(m)} = \{dep(w_s, w_t) | w_s, w_t \in W^{(m)}\} \tag{2}$$

Definition 5 (Mashup Functionality Annotation). A functional description $W^{(m)}$ of a mashup service M corresponds to a markup description $SW^{(m)}$, expressed by

$$SW^{(m)} = \{\langle w_1, \{api_{11}, \cdots\}\rangle, \langle w_2, \{api_{21}, \cdots\}\rangle, \cdots\} \tag{3}$$

Where $\{w_1, w_2, \cdots\}$ are words in the description of $W^{(m)}$. $\{api_{k1}, \cdots\}$ states the set of APIs annotated by w_k.

Definition 6 (Business Execution Processes Extraction). Given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, the task of extracting business execution processes of APIs in M is defined as

$$g(SW^{(m)}, C^{(m)}) :\ = G' \tag{4}$$

Where $SW^{(m)}$ is a semantically annotated functional description of $W^{(m)}$; $C^{(m)}$ is the set of atomic grammatical dependencies in $W^{(m)}$; g is an effective approach that derives a generated graph G', corresponding to the desired business execution processes of APIs in M.

It is observed that given a mashup service M or a mashup functional description with a set of recommended APIs, we mainly focus on how to semantically annotate its APIs from $W^{(m)}$ to $SW^{(m)}$ and design an effective approach g for business execution processes extraction.

3 Automated Extraction of Business Execution Processes

3.1 Framework of the Approach

Figure 1 illustrates the overall framework of our proposed approach. From the perspective of task functionality, it goes through three crucial stages, including word-domain model construction, mashup functionality API annotation, and dependency network extraction.

In the stage of word-domain model construction, all the mashup services are aggregated to form a word bank. Domains are derived from the partitioning of original API and mashup service repository. WordNet is then applied to calculate the semantic similarity degree of a pair of word and domain. In the stage of mashup functionality API annotation, the similarity between a word and an

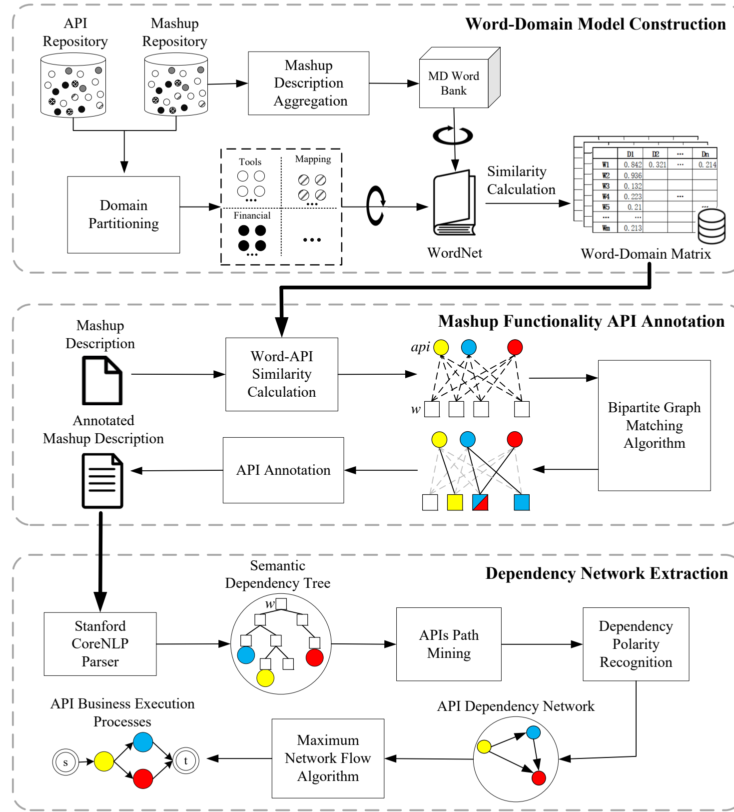

Fig. 1. The framework of our approach.

API is evaluated by word-domain matrix and API-domain concurrence matrix. Furthermore, we transform a mashup annotation problem to a bipartite graph and adopt the maximum bipartite graph matching algorithm to annotate APIs in mashup service. In the stage of dependency network extraction, we generate a set of semantic relationship trees for an annotated mashup description via Stanford CoreNLP parser. By evaluating the polarity of invocation relationships among APIs based on analyzing path dependencies, we construct an API dependency network where network maximum flow algorithm is applied to detect start and end points for business execution processes extraction.

3.2 Word-Domain Model Construction

To best match feasible words for an API in a mashup service, the similarity calculation between them becomes a key factor. Since domain and API concurrency matrix can be available from service description, word and API relationships can be directly derived if we have word and domain matrix model.

Definition 7 (Word-Domain Matrix). The word-domain model is denoted by an $m * n$ matrix M_{d_w}, where m is the number of all service domains and n is the number of words collected from all mashup functional descriptions.

$$M_{d_w} = \begin{array}{c} \\ d_1 \\ d_2 \\ \vdots \\ d_m \end{array} \begin{array}{c} \begin{array}{ccc} w_1 & w_2 \ldots & w_n \end{array} \\ \begin{pmatrix} sem_{11} & sem_{12} & \ldots & sem_{1n} \\ sem_{21} & sem_{22} & \ldots & sem_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ sem_{m1} & sem_{m2} & \ldots & sem_{mn} \end{pmatrix} \end{array} \tag{5}$$

Each entry sem_{ij} represents the semantic similarity degree between d_i and w_j, ranging from 0 to 1. All the domains $D = \{d_1, \ldots, d_m\}$ are accumulated by domain partitioning from API and mashup service repository, while all the words $W = \{w_1, \ldots, w_n\}$ are collected and preprocessed from mashup service repository.

We use WordNet as a lexical database to measure the semantic similarity between domain and word. Domains and mashup functional description words are mapped into a hierarchical semantic tree in WordNet representation, where the similarity degree of two nodes can be calculated based on their path distance. Given a domain d_i and a mashup functional description word w_j, the semantic similarity degree sem_{ij} is calculated by

$$sim\left(d_i, w_j\right) = \frac{2 * depth\left(lso\left(d_i, w_j\right)\right)}{len\left(d_i, w_j\right) + 2 * depth\left(lso\left(d_i, w_j\right)\right)} \tag{6}$$

Where, $lso\left(d_i, w_j\right)$ represents the deepest common parent between d_i and w_j. $depth\left(lso\left(d_i, w_j\right)\right)$ is the depth of $lso\left(d_i, w_j\right)$. $len\left(d_i, w_j\right)$ represents the shortest path length between d_i and w_j.

Due to polysemy, a mashup functional description word may correspond to multiple concepts in semantic dictionary. In order to eliminate the ambiguity between two words, an improved similarity calculation algorithm is used to maximize the semantic matching degree. Given a domain d_i and a word w_j, the updated semantic similarity degree sem_{ij} is calculated by

$$sim'\left(d_i, w_j\right) = \max_{c_x \in synsets(d_i), c_y \in synsets(w_j)} sim(c_x, c_y) \tag{7}$$

Where, $synsets\left(d_i\right)$ and $synsets\left(w_j\right)$ represent the collection of concepts corresponding to d_i and w'_j respectively.

3.3 Mashup Functionality API Annotation

In this section, we describe how to map the words in a mashup functionality description to the corresponding involved APIs. Given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, API annotation is to match the most mashup appropriate functional description words for each API that belongs to M. It converts original mashup functional description $W^{(m)}$ to the annotated one $SW^{(m)}$, which is formally expressed by

$$f(W^{(m)}) = SW^{(m)} \tag{8}$$

$$\begin{gathered} W^{(m)} = \{w_1, w_2, \cdots\} \\ \Downarrow f \\ SW^{(m)} = \{\langle w_1, \{api_{11}, \cdots\}\rangle, \langle w_2, \{api_{21}, \cdots\}\rangle, \cdots\} \end{gathered} \tag{9}$$

Where function f performs two steps, including the task of semantic similarity calculation between API and word (f_a) and the task of optimally matching an API to its semantically correlative words (f_b). Thus, mashup functionality API annotation can be decomposed as

$$f\left(W^{(m)}\right) = f_b\left(f_a\left(W^{(m)}\right)\right) \tag{10}$$

(1) f_a: semantic similarity calculation between API and word. Derived from API service repository, the concurrency tagging between an API api_i and a domain d_j indicates that whether api_i belongs to d_j. Here, another API-domain matrix denoted as M_{a_d} reflects the relationship.

$$M_{a_d} = \begin{array}{c} \\ api_1 \\ api_2 \\ \vdots \\ api_l \end{array} \begin{array}{c} \begin{array}{cccc} d_1 & d_2 & \dots & d_m \end{array} \\ \begin{pmatrix} tag_{11} & tag_{12} & \dots & tag_{1m} \\ tag_{21} & tag_{22} & \dots & tag_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ tag_{l1} & tag_{l2} & \dots & tag_{lm} \end{pmatrix} \end{array} \tag{11}$$

Where each row and column in the matrix represents an API and a domain, respectively. Each entry is either equal to 0 or 1. If the domain tagging of api_i is marked by d_j, tag_{ij} equals 1, while the rest of the values in M_{a_d} are 0.

Based on the API-domain matrix M_{a_d} and constructed word-domain matrix M_{d_w}, the weighting matrix M_{a_w} between API and mashup functional description word can be directly produced with the multiplication of M_{a_d} and M_{d_w}.

$$M_{a_w} = \begin{array}{c} \\ api_1 \\ api_2 \\ \vdots \\ api_l \end{array} \begin{array}{c} \begin{array}{cccc} w_1 & w_2 & \dots & w_n \end{array} \\ \begin{pmatrix} wt_{11} & wt_{12} & \dots & wt_{1n} \\ wt_{21} & wt_{22} & \dots & wt_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ wt_{l1} & wt_{l2} & \dots & wt_{ln} \end{pmatrix} \end{array} \tag{12}$$

Taking the above matrices, given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)}\rangle$, the semantic similarity degree between an API and a mashup functional description word can be calculated. Formally, two submatrices $M_{a_d}^{(m)}$ and $M_{d_w}^{(m)}$ are intercepted to deduce semantic similarity matrix $Sim_{a_w}(m)$.

$$Sim_{a_w}(m) = M_{a_d}^{(m)} \times M_{d_w}^{(m)} \tag{13}$$

Where $M_{a_d}^{(m)}$ represents a submatrix of M_{a_d} whose rows are composed of APIs involved in $L^{(m)}$. Similarly, $M_{d_w}^{(m)}$ represents a submatrix of M_{d_w} whose

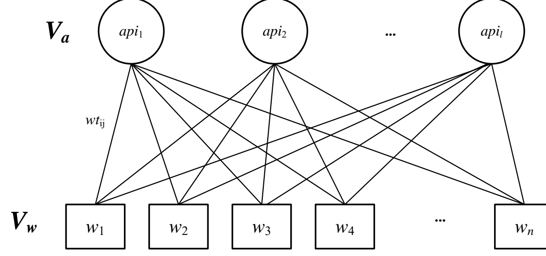

Fig. 2. The transformed weighted bipartite graph for API annotation.

columns are composed of words involved in $W^{(m)}$. As a result, $Sim_{a_w}(m)$ is a submatrix of M_{a_w} and each entry reflects the weighting between an API and a word in M.

(2) f_b: optimally matching an API to its semantically correlative words. By applying the generated similarity matrix $Sim_{a_w}(m)$, API annotation problem for a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$ is transformed to a fully connected and weighted bipartite graph $G = \langle V, E \rangle$. Specifically, API vertices $V_a \subset V$ originates from all of the APIs in $L^{(m)}$; word vertices $V_w \subset V$ originates from all of the mashup functional description words in $W^{(m)}$; the weighting wt_{ij} of each edge in E corresponds to an entry in $Sim_{a_w}(m)$. The transformed bipartite graph for API annotation of mashup functionality is illustrated in Fig. 2.

In this way, the solution to a mashup functionality API annotation problem is formally equivalent to best finding a partition of the bipartite graph G. Here, we apply a threshold-based API annotation algorithm called T-WDM and an API annotation algorithm called G-WDM based on bipartite graph maximum matching. For the T-WDM algorithm, if an edge weighting wt_{ij} in G is greater than predefined threshold θ, then api_i is annotated by w_j. Based on bipartite graph maximum matching, G-WDM is an improvement of Kuhn-Munkres method, which optimizes the matching between APIs and words.

Note that API annotation is a content-based matching problem like LDA topic modeling, which can be thought as classifying each word into a category of APIs. Since the words that actually express the API has low frequency in the entire large-scale service repository, it can easily occur inaccuracy with high error on mapping words to API tags using LDA modeling. To avoid this application scenario, we leverage an external lexical database WordNet for similarity calculation between mashup description words and API tags.

From the maximum matching solution to the bipartite graph, we mark the original mashup functional description $W^{(m)}$ to its semantically annotated one $SW^{(m)}$, by replacing the words with their matched APIs.

3.4 Dependency Network Extraction

In this section, we generate API business execution processes based on dependency network, which can be extracted by parsing mashup description and analyzing the path dependency relationships.

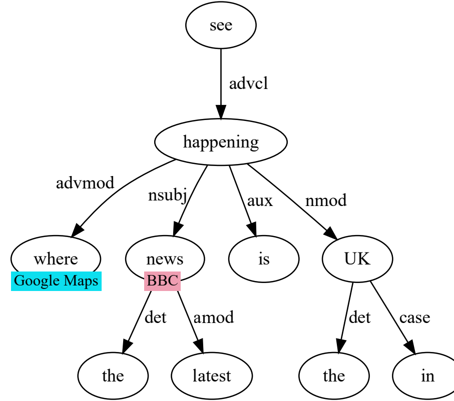

Fig. 3. Semantic dependency tree is extracted from a mashup functionality description. CoreNLP tool is used to analyze the functionality description and derive its corresponding semantic dependency tree of a mashup service. In the figure, those words in the eclipse come from mashup service description; those words around the arrows represent the dependency relationships; those words in the rectangles below the circles are tags marked for involved APIs.

Given an annotated mashup service description $SW^{(m)}$, Stanford CoreNLP parser [2] is applied to further generate a set of semantic dependency trees, each of which represents a sentence in $SW^{(m)}$. An edge in a semantic dependency tree states a unique binary relationship by a dependency type $r\left(w_a, w_b\right)$. r represents a specific dependency type; w_a and w_b represent a dominant word and a dependent word in the relationship, respectively. Taking an annotated service description $SW^{(m)}$ as input, we mark an API to its corresponding position in semantic dependency tree. For example, given a mashup functional description *"See where the latest news is happening in the UK"*, its semantic dependency tree is illustrated in Fig. 3.

As shown in Fig. 3, there are multiple dependency types among words in a semantic dependency tree. The binary dependency modification between two words for a dependency type is defined as below.

Definition 8 (Binary Dependency Modification). Given a dependency type $r\left(w_a, w_b\right)$, the binary dependency modification between w_a and w_b holds three different kinds of possibilities:

(1) If w_a directly modifies w_b on r, it is denoted as $w_a \overset{r}{\Rightarrow} w_b$;
(2) If w_a is reversely modified by w_b on r, it is denoted as $w_a \overset{r}{\Leftarrow} w_b$;
(3) If w_a and w_b are mutually modified on r, it is denoted as $w_a \overset{r}{\Leftrightarrow} w_b$.

Based on above definition, all dependency types are classified into three categories, including ***positive***, ***negative*** and ***neutral***.

Definition 9 (Dependency Relationship Polarity). Given a dependency type $r\left(w_a, w_b\right)$, its dependency relationship polarity can be identified by a piecewise function $H\left(r\right)$

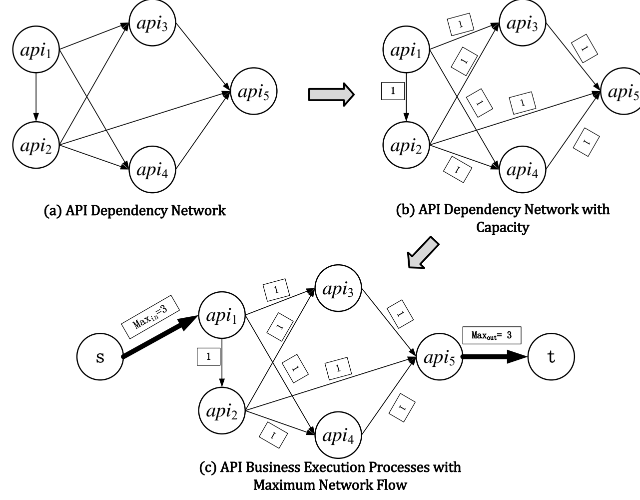

Fig. 4. Extracted API dependency network and business execution processes.

$$H(r(w_a, w_b)) = \begin{cases} positive, & \text{if } w_a \overset{r}{\Rightarrow} w_b \text{ is satisfied} \\ neutral, & \text{if } w_a \overset{r}{\Leftrightarrow} w_b \text{ is satisfied} \\ negative, & \text{if } w_a \overset{r}{\Leftarrow} w_b \text{ is satisfied} \end{cases} \quad (14)$$

By using $H(r)$, dependency polarity of two APIs in a semantic dependency tree can be recognized via multiplicative chain rule. Given two APIs api_i and api_j, assume that it has a reachable path from api_i to api_j, denoted as $dep(api_i, api_j) = \{api_i \xrightarrow{td_{z1}} w_{z1} \xleftarrow{td_{z2}} w_{z2} \xrightarrow{td_{z3}} \ldots \xrightarrow{td_{zn}} api_j\}$, the dependency polarity of these two APIs is calculated by

$$L(dep(api_i, api_j)) = \prod_{(w_i, w_j)} \delta_{(w_i, w_j)} \cdot H(r(w_i, w_j)), \quad (w_i, w_j) \in dep(api_i, api_j) \quad (15)$$

Where $\delta_{(w_i, w_j)}$ is a symbol term indicating the dependency direction in the pathway chain of $dep(api_i, api_j)$. $H(r(w_i, w_j))$ is the dependency relationship polarity for a dependency type $r(w_i, w_j)$ in $dep(api_i, api_j)$.

Performing the calculation of dependency polarity between every pair of APIs in a mashup service, we extract an API dependency network.

Definition 10 (API Dependency Network). Given a mashup service $M = \langle W^{(m)}, L^{(m)}, D^{(m)} \rangle$, its API dependency network is a directed graph $G' = \langle V', E' \rangle$. $V' = L^{(m)}$ and an edge $e' \in E'$ satisfies the conditions:

(1) $E' = \{e' | e' = \langle api_i, api_j \rangle, api_i, api_j \in L^{(m)}\}$;
(2) The dependency polarity of $L(dep(api_i, api_j))$ is positive.

When extracting API dependency network, if the two APIs (api_i, api_j) has positive dependency polarity, an edge $e' = \langle api_i, api_j \rangle$ is added to E'; if it has negative dependency polarity, an edge $e' = \langle api_j, api_i \rangle$ is added to E'; if it has

neutral dependency polarity, above two edges are both added to E'. An API dependency network is shown in Fig. 4(a).

As shown in Fig. 4(a), the extracted API dependency network G' can be regarded as a candidate of business execution processes before a starting and ending point are chosen, respectively. To pick them out, the capacity of each edge in G' is set as 1 as shown in Fig. 4(b). Under this setting, the maximum network flow algorithm [3] is applied to check the entire network traffic value of G'. When it reaches the maximum traffic value, the reliance level arrives at the optimum state. The starting and ending points of the maximum flow corresponds to the ones where the sum of in-degree and the sum of out-degree are the maximum. Consequently, business execution processes of APIs in a mashup service can be extracted as shown in Fig. 4(c).

4 Experiments

4.1 Experimental Setup and Dataset

We developed a prototype system and all modules are implemented in Java. It integrates WordNet[1] for word-domain matrix construction and Stanford CoreNLP parser for API dependency network extraction. Meanwhile, four competing approaches are integrated in our prototype system. All the experiments have been carried out on a PC with an Intel Dual Core 2.8 GHz processor and 4GB RAM in Windows 10.

The dataset used in the experiment was collected from the largest online service management platform ProgrammableWeb.com[2]. As of May 2018, ProgrammableWeb recorded more than 19,000 API services and approximately 8,000 mashup services. We crawled the information of APIs and mashup services, including name, functional description and domain. After the preprocessing, we obtained a collection of 13,869 API services and 6,254 mashup services. The statistics of experimental dataset is shown in Table 1.

Table 1. The statistics of experimental dataset crawled from ProgrammableWeb.

Dataset Item	Value
Total number of mashup services	6,254
Total number of API services	13,869
Total number of domains	474
Average number of mashups in a domain	13.927
Average number of APIs in a domain	2.121
Total number of words in mashup descriptions	112,987

All the mashup services, including two or more than two API services are selected from Table 1, to conduct the experiments. We classify these mashups based on the number of API that mashup contains. Each class randomly extracted 100

[1] https://wordnet.princeton.edu/
[2] https://www.programmableweb.com

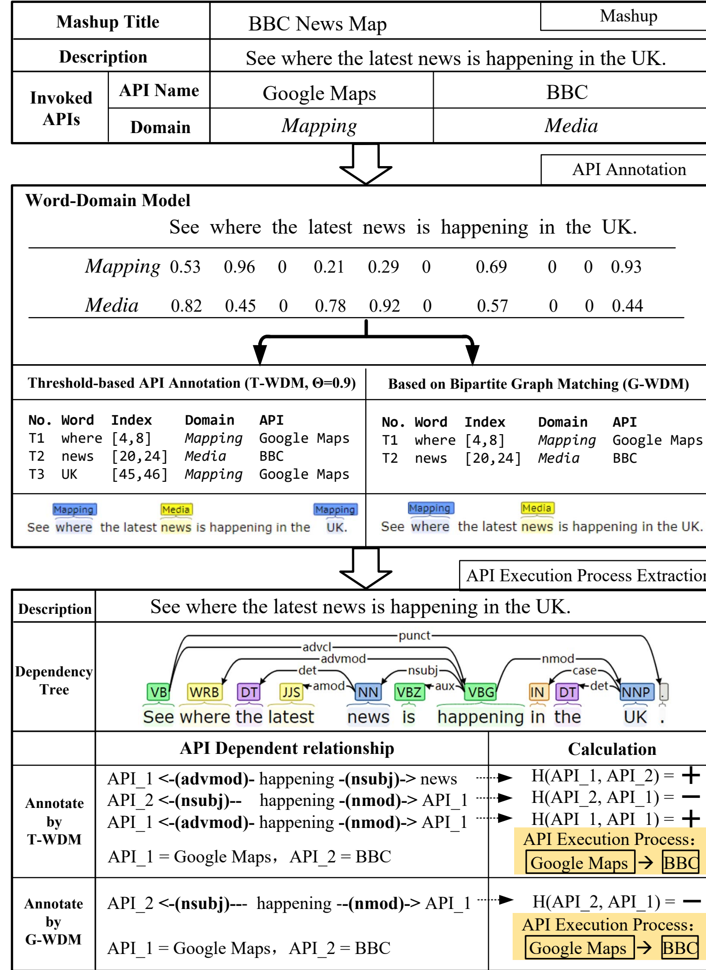

Fig. 5. The experimental result of extracting API business execution processes from a mashup service requirement.

samples as experimental data. Because the data on the ProgrammableWeb does not contain the API execution process sequence, we invite experienced web development experts to manually determine the execution process of each mashup service. For each mashup record we only keep the mashup identification number (including), natural language text description, API list and domain. The API set as experimental data is directly using all the APIs that we collected. For each API record we keep the API ID, natural language text description and domain.

4.2 Case Study of Business Execution Processes Extraction

To validate the applicability, we show a case study of three crucial stages for API business execution processes. Fig. 5 illustrates a specific case of API business

execution processes extraction for mashup *"BBC News Map"*. Its functional description is described as *"See where the latest news is happening in the UK"*. Two involved APIs are invoked for the implementation of mashup functionality, including *Google Maps* in domain *Mapping* and *BBC* in domain *Media*. There are 10 tokens in this description except punctuation. After removing the nonsensical prepositions and particles (*the*, *is*, and *in*), there are 6 remaining tokens. After the preprocessing, it goes through three steps as below.

(1) The word-domain matrix is constructed to evaluate the similarity between 6 words and 2 domains. The generated word-domain matrix is shown in the second part of Fig. 5.
(2) For API annotation, we can find the most similar word to the first domain *Mapping* from the word-domain matrix. Suppose we use the constant-threshold method (T-WDM) method. Under the threshold 0.9, we can find the two words of *where* (Similarly 0.97) and *UK* (0.93). For another domain *media*, we can find a word *news* (0.97). In this way, we will give the corresponding API tags to these words we find.
(3) In the dependency network extraction, we invoke Stanford CoreNLP to analyze mashup functionality description and obtain a parser tree, where it consists of 10 basic Stanford dependency types. In these parse result, related to the tagged APIs is *happening* $\rightarrow$ *where* = *advmod*(negative), and *happening* $\rightarrow$ *news* = *nsubj*(positive). Using the proposed multiplicative chain rule, we can calculate the dependency relationship polarity: $where \rightarrow news = positive$, which is $H\left(Google\ Maps, BBC\right) = positive$. Calculate all tagged API pairs in this method, then we can extract API execution business processes: $Google\ Maps \Rightarrow BBC$.

4.3 Competitive Methods and Evaluation Metrics

To demonstrate the effectiveness of our approach for API business execution processes extraction, we compare our two self-developed methods with two baseline ones. Here, the differences among these competing methods lie in API annotation.

(1) LED [7]: Levenshtein edit distance is an annotation algorithm based on the edit distance similarity calculation. This method measures the minimum number of edit operations required to convert from one string to another.
(2) STEM: It is an annotation algorithm based on stem similarity calculation. This method extracts stems from two words and calculates their similarity degree using string comparison strategy.
(3) T-WDM: This method constructs a word-domain matrix for semantic similarity degree calculation, and then transforms API annotation problem to a bipartite graph model. It identifies those edges that reach or exceed the threshold for semantic annotation of APIs in a mashup service.
(4) G-WDM: Based on T-WDM, this method annotates API service by applying the maximum matching in weighted bipartite graph. In light of the mapping among words and APIs, it performs API annotation in a mashup service.

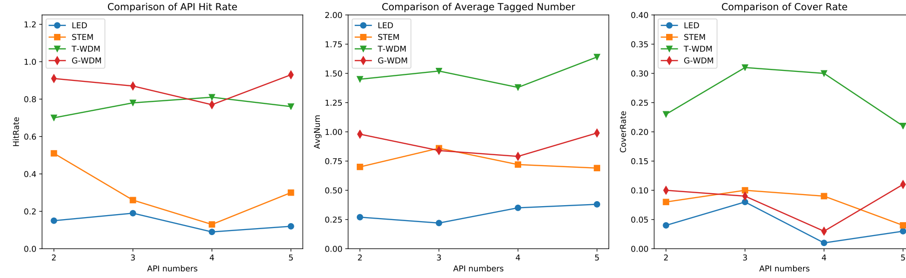

Fig. 6. Experimental results on API annotation among four competing approaches.

We evaluate the effectiveness of above approaches in terms of two aspects, including API annotation and business execution processes extraction in mashup service. To test the performance of API annotation, three evaluation metrics are used including API hit rate $HitRate$, API average tagged times $AvgTag$ and API cover rate $CoverRate$. $HitRate$ is defined as the ratio of the sum of annotated APIs and the total number of APIs involved in a mashup service. $AvgTag$ is defined to measure the average number of times that an API was tagged in a mashup service. $CoverRate$ is defined as the ratio of the sum of words used to annotate more than one API service and the total number of words used to annotate APIs in a mashup service.

To test the performance of business execution processes extraction in mashup, $Recall$, $Precision$ and $F\!-\!measure$ are used in the experiments. $Precision$ is defined as the ratio of the number of correctly extracted business execution processes and the total number of extracted business execution processes. $Recall$ is defined as the ratio of the number of mashup services where business execution processes are accurately extracted and the total number of mashup services. $F\!-\!measure$ is a comprehensive indicator based on $Precision$ and $Recall$.

4.4 Comparative Results and Analyses

In API annotation experiments, we test the performance of four competing methods on the evaluation metrics $HitRate$, $AvgTag$ and $CoverRate$. The experimental results on API annotation are shown in Fig. 6.

The horizontal axis represents the number of APIs invoked by a mashup service and the vertical axis represents the performance on each evaluation metric. Overall, three competing approaches based on semantic similarity calculation (T-WDM, G-WDM and STEM) are better than LED on all three-evaluation metrics. The main reason is that the style of mashup functional description words has many variants, while LED does not take into account semantics and can only establish logical connections among mashup and API services that use the same vocabulary.

More specifically, our two proposed approaches G-WDM and T-WDM outperform STEM and LED on $HitRate$, because latent topics have been applied to calculate semantic similarity degree based on WordNet. Regarding $AvgTag$, our approach T-WDM reaches the highest performance, while G-WDM is supe-

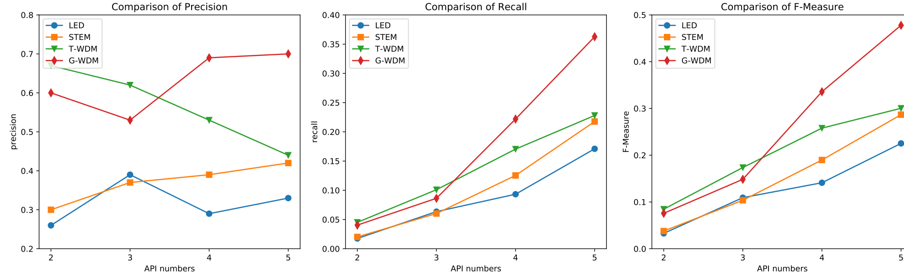

Fig. 7. Experimental results on business execution processes extraction.

rior to another two competing approaches. The main reason is that a bunch of words can be matched with an API when the threshold is set as a small value in T-WDM, while an API can be approximately annotated by a word in G-WDM by the maximum matching of bipartite graph. Conversely, low similarity degree leads to a smaller number of matched words on average for an API in STEM and LED. As for *CoverRate*, our approach T-WDM has the highest value, while other three competing approaches are much lower than T-WDM. The underlying reason is that they either can only match approximately a mashup functional description word for an API in G-WDM or less than that in STEM and LED.

In the experiment of extracting business execution processes from a mashup, we compare the performance of the four competing approaches on *precision*, *recall* and $F-measure$. The experimental results are shown in Fig. 7.

From the experimental results in Fig. 7, we conclude that with the increase of the number of APIs invoked by a mashup service, our two proposed approaches T-WDM and G-WDM are better than STEM and LED. Due to the loosely relational selection strategy, the precision of T-WDM becomes smaller as the increasing number of APIs involved in a mashup service. On the contrary, the precision of G-WDM becomes bigger as the number of APIs increases in a mashup service, counting on the effective annotation by bipartite graph maximum matching. Generally, the precision of T-WDM and G-WDM exceeds more than 50% no matter how the number of APIs varies in a mashup service, whereas STEM and LED are both less than 45%. In conclusion, the experimental results validate the feasibility and effectiveness of our proposed approach.

4.5 Performance Impact of Parameters

The proposed approach T-WDM takes a threshold θ as the constraint during API annotations. This parameter directly affects the effectiveness of API annotation. In order to test its influence, we set the value of θ from 0.7 to 0.9 with a step size of 0.1 and compare the performance with G-WDM on *HitRate*, *AvgTag* and *CoverRate*. The experimental results are illustrated in Fig. 8.

It can be observed that the effectiveness of T-WDM becomes worse along with the increasing number of θ. The explanation is that as the threshold of semantic similarity degree increases, the number of matched words for an API obtained by T-WDM decreases.

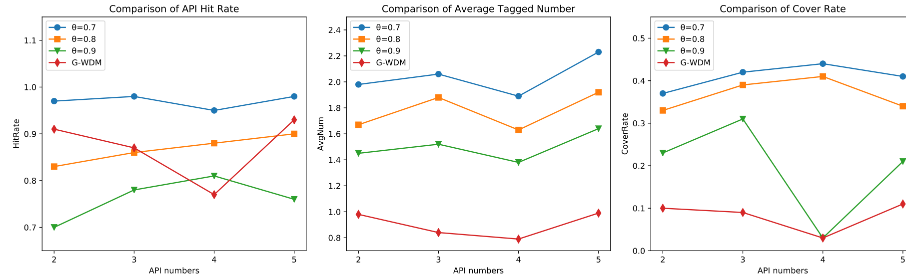

Fig. 8. Experimental results on parameter impact of API annotation.

For our proposed two approaches, T-WDM is superior to G-WDM in terms of $AvgTag$ and $CoverRate$, regardless of the variations of θ in the experimental setting. Note that once a bipartite graph is modeled for API annotation, the results by G-WDM keep unchanged on three evaluation metrics, since the mapping from an API to its annotated words are found by the maximum matching algorithm. Therefore, when θ becomes large enough, the values of $AvgTag$ and $CoverRate$ in T-WDM could be lower than that in G-WDM. The reason is that the value of these two metrics tends to be 0, if the threshold is so large that no appropriate words can be matched for an API.

5 Related Work

In this section, we mainly review the recent advancements on service recommendation and mashup creation that is highly related with our work.

To assist mashup developer's API selection and improve the efficiency of mashup creation, the authors in [1,11] extracted user's social attribute and interests' portrait to recommend candidate APIs for mashup service creation. The authors in [14] proposed a dynamic mashup recommendation system, where service evolution has been taken into account by exploiting LDA topic model and time series prediction. The authors in [10] enriched service recommendation results for mashup creation by employing variant K-means. This can enhance service categorization and restrain candidate services from each category.

Recently, the authors in [9] proposed an API recommendation method for mashup development using matrix factorization where mashup services are clustered through a two-level topic model. The authors in [12] proposed a new technique to fast and accurately build an API network using semantic similarity construction and community detection. By doing so, mashup developers are freed from the exhausting search phase to find their desired APIs. The authors in [4] proposed a novel service set recommendation framework for mashup creation by applying an improved clustering algorithm vKmeans and hypergraph modeling. It solves the problem of redundant service recommendation and ignored cooperation relations among services. The authors in [13] proposed a probabilistic matrix factorization approach to discover implicit co-invocation patterns between APIs, finding more accurate API rankings for mashup recommendation.

From the above investigation, we observe that the existing methods mainly focus on how to accurately and efficiently find appropriate APIs for mashup creation. Due to lacking business execution processes of these recommended APIs, it is still a challenging and time-consuming task for software developers to create a new mashup service.

6 Conclusion

In this paper, we propose a novel approach to extract API business execution processes for mashup creation. It goes through three crucial stages including word-domain model construction, mashup description API annotation and dependency network extraction. Extensive experiments conducted on large-scale real-world APIs and mashup services crawled from ProgrammableWeb validate the feasibility and effectiveness of the proposed approach. By providing the invocation relationships among APIs, it could be potentially applied to assist software developers in expediting the procedure of mashup creation. In the future work, we will consider using a greater number of criteria, including API functionality, reliability, and a range of non-functional features such as cost, performance, and API provider reputation, when recommending a set of web APIs with business execution processes.

Acknowledgement

This work was partially supported by Shanghai Natural Science Foundation (No. 18ZR1414400 and 17ZR1400200), National Natural Science Foundation of China (No. 61772128 and 61303096), Shanghai Sailing Program (No. 16YF1400300), and Fundamental Research Funds for the Central Universities (No. 16D111208).

References

1. Cao, B., Liu, J., Tang, M., Zheng, Z., Wang, G.: Mashup service recommendation based on user interest and social network. In: IEEE International Conference on Web Services (ICWS). pp. 99–106. IEEE (2013)
2. De Marneffe, M.C., Manning, C.D.: Stanford typed dependencies manual. Tech. rep., Stanford University (2008)
3. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. In: Combinatorial Optimization—Eureka, You Shrink!, pp. 31–33. Springer (2003)
4. Gao, W., Wu, J.: A novel framework for service set recommendation in mashup creation. In: IEEE International Conference on Web Services (ICWS). pp. 65–72. IEEE (2017)
5. Gao, Z., Fan, Y., Wu, C., Tan, W., Zhang, J., Ni, Y., Bai, B., Chen, S.: SeCo-LDA: mining service co-occurrence topics for recommendation. In: IEEE International Conference on Web Services (ICWS). pp. 25–32. IEEE (2016)

6. Jain, A., Liu, X., Yu, Q.: Aggregating functionality, use history, and popularity of APIs to recommend mashup creation. In: International Conference on Service-Oriented Computing (ICSOC). pp. 188–202. Springer (2015)
7. Levenshtein, V.: Binary codes capable of correcting spurious insertions and deletion of ones. Problems of information Transmission **1**(1), 8–17 (1965)
8. Li, C., Zhang, R., Huai, J., Sun, H.: A novel approach for API recommendation in mashup development. In: IEEE International Conference on Web Services (ICWS). pp. 289–296. IEEE (2014)
9. Rahman, M.M., Liu, X., Cao, B.: Web API recommendation for mashup development using matrix factorization on integrated content and network-based service clustering. In: IEEE International Conference on Services Computing (SCC). pp. 225–232. IEEE (2017)
10. Xia, B., Fan, Y., Tan, W., Huang, K., Zhang, J., Wu, C.: Category-aware API clustering and distributed recommendation for automatic mashup creation. IEEE Transactions on Services Computing **8**(5), 674–687 (2015)
11. Xu, W., Cao, J., Hu, L., Wang, J., Li, M.: A social-aware service recommendation approach for mashup creation. In: IEEE International Conference on Web Services (ICWS). pp. 107–114. IEEE (2013)
12. Yang, X., Cao, J.: A fast and accurate way for API network construction based on semantic similarity and community detection. In: IFIP International Conference on Network and Parallel Computing (NPC). pp. 75–86. Springer (2017)
13. Yao, L., Wang, X., Sheng, Q.Z., Benatallah, B., Huang, C.: Mashup recommendation by regularizing matrix factorization with API co-invocations. IEEE Transactions on Services Computing (2018), https://doi.org/10.1109/TSC.2018.2803171
14. Zhong, Y., Fan, Y., Huang, K., Tan, W., Zhang, J.: Time-aware service recommendation for mashup creation. IEEE Transactions on Services Computing **8**(3), 356–368 (2015)