Code Search via Topic-Enriched Dependence Graph Matching

Shaowei Wang, David Lo, and Lingxiao Jiang
School of Information Systems
Singapore Management University
Email: {shaoweiwang.2010,davidlo,lxjiang}@smu.edu.sg

***Abstract*—Source code contains textual, structural, and semantic information, which can all be leveraged for effective search. Some studies have proposed semantic code search where users can specify query topics in a natural language. Other studies can search through system dependence graphs. In this paper, we propose a semantic dependence search engine that integrates both kinds of techniques and can retrieve code snippets based on expressive user queries describing both topics and dependencies. Users can specify their search targets in a free form format describing desired topics (*i.e.*, high-level semantic or functionality of the target code); a specialized graph query language allows users to describe low-level data and control dependencies in code and thus helps to refine the queries described in the free format. Our empirical evaluation on a number of software maintenance tasks shows that our search engine can efficiently locate desired code fragments accurately.**

***Keywords*-Code search; topic modelling; dependence graphs;**

I. Introduction

Programmers spend a lot of time in code search, searching for not only relevant program components, functions, but also specific code snippets. To find relevant code fragments, a developer may employ a search utility that comes with their operating systems (*e.g.*, Windows Explorer) or IDEs (*e.g.*, Eclipse, Visual Studio). However, the functionality of these tools is often limited. First, they can only provide exact matching of strings or regular expressions, which require a user to provide very specific queries. Second, code is texts with structures and semantics; such information is implicit and these string-based search tools are unable to capture dependence relations among program elements. Third, intended search targets by a user can involve code snippets spread across multiple different program locations; it is difficult for these tools to capture relevant non-consecutive code snippets.

To address such limitations, one kind of approaches is to use natural language processing techniques to find relevant code [5], [9]. Other studies enable users to specify rich queries involving dependencies among program elements of interest [4], [14]. Both kinds of studies have their disadvantages. Natural language processing techniques cannot easily specify fine-grained relations among particular elements in the code, while dependence based search often fails to distinguish code of different high-level concepts but with similar low-level relationship. As the result, both kinds of techniques may produce various false positives during search.

In this work, we marry the use of natural language and graph query processing to realize a code search engine that supports expressive queries allowing natural language texts and dependence relations. We also improve the speed of current state-of-the-art transitive dependence code search engine tool proposed in [14] by an order of magnitude speedup on average.

Our solution incorporates a topic modeling engine based on Latent Dirichlet Allocation (LDA) [11] that learns topics from and assigns them to documents. We also incorporate recent advances in querying labeled graphs [7] to alleviate the limitation of code search engine. Our framework first takes code bases and extracts structural and semantic representations (*i.e.*, system dependence graphs (SDGs) and topics) from the code. The topics are assigned to the various nodes in the SDG to create an enriched SDG (eSDG). Finally, these eSDGs are used by our graph query engine to answer user queries that are expressed in our query language in which a user can specify both free form texts and complex dependence relations.

II. Related Work

General-Purpose Code Search. General-purpose code search tools allow users to specify queries in various forms and return matching code. Some tools allows queries in plain text [5], [8], [10]; others allow queries describing program dependencies [14]. We extend these studies by providing users the combined capability to handle both plain texts and dependence queries, and leverage a topic modeling engine and a graph reachability engine to significantly improve the efficiency of dependence constraints processing. Our code search is also different from Sourcerer [8], Exemplar [3], and Portfolio [10] in that we focus on searching for code snippets instead of applications, components, or functions.

Specialized Code Search. Some other techniques could be categorized as a form of specialized code search: code clone detection [1], [6], recommendation of code samples [12], [15], etc. Different form the above studies, we provide a general-purpose code search tool supporting: textual constraints, dependence constraints, and topic modeling.

III. Proposed Framework

Our approach is composed of code processing, text processing, graph enrichment, and query processing. The relations among these components are shown in Figure 1.

Code and Text Processor Our code processor component takes in source code and outputs its corresponding System

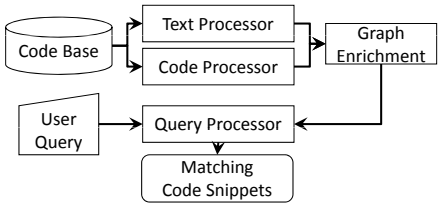

Fig. 1. Our Code Search Overview.

Dependence Graph (SDG). We use CodeSurfer to generate SDGs [2]. The nodes in the SDGs are labeled with different program element types, and edges between nodes represent control or data dependencies.

Our text processor component takes in source code and outputs a topic model. The processor first changes each method in a code base into a bag of words. For each method, we take the name of the method, comments and identifiers in the method, split some names into word tokens, and remove some keywords and non-interesting words to form the bag of words. Each bag of words forms a document. A code base is thus transformed into a set of documents. These documents are then fed into LDA [11] to produce 50 different topics where each topic is represented by a set of words.

Graph Enrichment Component. This component takes in the SDGs outputted by the code processor and the topics outputted by the text processor, and enriches the SDG nodes with the topics. After the enrichment process, each SDG node contains not only the the type of the program element associated with it, but also, a set of corresponding topics with their associated probabilities. For example, a node in the SDG could be associated with topic-1 with probability 0.4, topic-2 with probability 0.3, and topic-3 with probability 0.3.And all nodes belonging to the same method would be associated with the same set of topics and their associated probabilities.

Query Processing Engine. The query processor is the key block in our proposed approach. It takes in a user query and, based on the enriched SDGs, returns all code snippets that match the query. Inside of the query processor, we first transform a user query in the format of DQL^t (*cf.* Section IV) into a graph representation, which is referred to as a *query graph* (*cf.* Section V-A). Then, we search through the enriched SDGs for program elements that match the query graph and return search results (*cf.* Section V-B). As this processor is rather complex, we describe it in the following two sections.

IV. Query Language

To help users formulate queries and provide inputs for our code search engine, we extend the Dependence Query Language (DQL) proposed in [14] with semantic topics. We refer to the new language as Dependence Query Language with Topic Modeling (DQL^T). Its syntax is shown in Table I.

DQL^T has five parts: topic declaration ($topic$), node declaration ($ndecl$), node description ($ndesc$), relationship description ($rdesc$), and targets ($target$). $Topic$ specifies the topics related to intended search results and can be any free form texts.

$query$	::=	($topic$)*; ($ndecl$)*; ($ndesc$)*; ($rdesc$)*; $target$;
$topic$	::=	$string$
$ndecl$	::=	$tlist$ id
$tlist$	::=	$tlist$ '|' $type$ | $type$
$type$	::=	$func$ | var | $assgn$ | $decl$ | $ctrlPoint$ | $stmt$
$ndesc$	::=	id ($cond$)*
$cond$	::=	[not] $ucond$ | similarTopicAs $string$
$ucond$	::=	contains $string$ | inFile $string$ | inFunc $string$ | atLine $number$ | ofType $string$ | ofControlType $ctype$
$ctype$	::=	for | while | switch | if
$rdesc$	::=	id op id
op	::=	[oneStep] $dependOp$ | $textOp$ | $structOp$
$dependOp$	::=	dataDepends | controls | calls
$textOp$	::=	contains
$structOp$	::=	isFieldOf | isElementOf
$target$	::=	(id)*
id	::=	$string$
$string$	::=	(A-Z,a-z,0-9)+
$number$	::=	(0-9)+

TABLE I
DQL^T Syntax

$Ndecl$ declares node variables and their types. $Ndesc$ specifies constraints on declared node variables. $Rdesc$ specifies constraints on the relations among declared node variables. $Target$ specifies the variables specified in $ndecl$ that are desired search targets. When a DQL^T query is processed on an enriched SDG, nodes in the eSDG that match the node variables specified in $target$ and satisfy the constraints specified in $ndecl$, $ndesc$ and $rdesc$ would be returned.

Node Declaration. This part of a query is to declare some node variables that will later be mapped to nodes in an eSDG. Each node variable can have one or more types (*i.e.*, a disjunction of types); each type can correspond to a kind of code in the code base (which is mostly in C) and is separated by '|' as defined by $tlist$. We consider six different types of node: $func$, var, $assgn$, $decl$, $ctrlPoint$, and $stmt$. A node of type $func$ corresponds to function invocations in the code. var corresponds to a variable. $assgn$ corresponds to an assignment. $decl$ corresponds to a variable or function declaration. $ctrlPoint$ corresponds to a branching condition that would affect code execution paths, *e.g.*, `if` or `while` conditions. $stmt$ corresponds to statements that would change control flows, *e.g.*, `return`, `break`, etc.

Node Description. This part of the query specifies further constraints on declared node variables ($cond$ and $ucond$). To specify constraints, developers can use the following unary operators: $contains$, $inFile$, $inFunc$, $atLine$, $ofType$, and $ofControlType$. The operator $contains$ allow developers to specify that a particular node needs to contain a particular text. The operators $inFile$ and $inFunc$ allow developers to specify a node that is located inside a particular file or function respectively. The operators $ofType$ and $ofControlType$ allow one to specify a node of a particular type or to be a control node of a particular type (i.e., for, while, switch, or if).

Different from DQL, DQL^T has a topic operator ($similarTopicAs$). With this operator one can specify the topics each node should belong to. Topic modeling makes it possible to use free form texts in the topic operator as topic modeling techniques can handle imprecise textual queries and relate different words that have similar meanings.

Relationship Description. This part of the query specifies constraints governing the relationships between two declared node variables. We have three types of relationships: dependence operator ($dependOp$), textual operator ($textOp$), and structural operator ($structOp$).

Dependence operators are used to specify either data dependence, control dependence, or transitive call relationship (i.e., there is a chain of function invocation from one node to another in the input SDG). They are expressed as operators: $dataDepends$, $controls$, and $calls$, respectively. Textual operator is used to specify that the textual content of one node contains (i.e., is a super string) of that of the other. Structural operator is used to specify that one node is a field of another ($isFieldOf$), or is an element of another ($isElementOf$). For example, "$a.b\ isFieldOf\ a$" holds, and "$a[b]\ isElementOf\ a$" also holds.

Targets. This part of a DQL^T query specifies the target node variables that would be returned as the output of the query. This set of variables is a subset of all declared variables. The declared variables would be matched to nodes in SDG, but only those specified as a target node variable would be returned. The other nodes serve as contexts for more accurate locating of the target nodes. Currently, the outputs of our code search engine are a set of line numbers linked to the locations of matching code fragments.

V. Query Processing Engine

Taking a user query and topic-enriched SDGs as input, our query processing engine goes through two major steps to produce search results: query graph construction and graph matching. Figure 2 illustrates the relations among the components in the query processing engine, which is a zoom-in of the `Query Processor` in Figure 1. Query graph construction converts a textual query into a graph representation. Graph matching locates nodes and edges in the input system dependence graphs that match the query graph and compose them into search answers. We describe them in the following sub-sections.

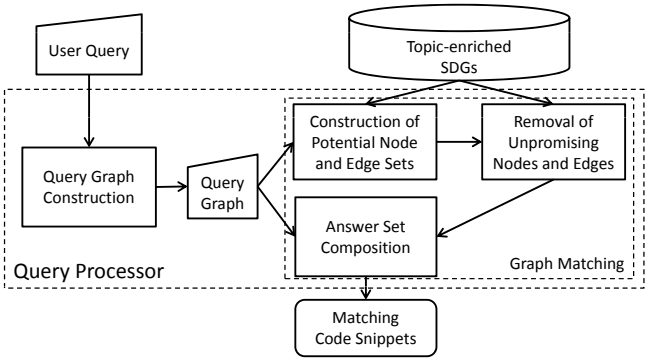

Fig. 2. Components in Query Processor.

A. Query Graph Construction

A user query can consist of node declarations, node descriptions, relationship descriptions, and the identifiers of the nodes of interest. We would form a graph from this textual description. Our DQL^T language syntax aims to make it intuitive for developers to specify their queries. The intuitive syntax may not directly correspond to the underlying system dependence graphs returned by CodeSurfer [2]. The purpose of the query graph construction step is to transform higher level DQL^T queries into a more low level query graph with nodes, edges, and their labels matching the SDG representations.

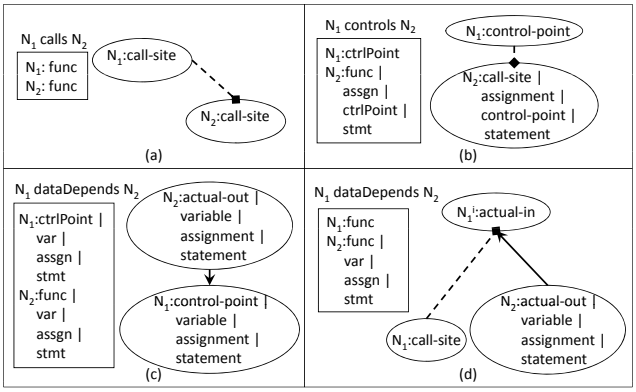

Fig. 3. Converting User Queries to Query Graphs. We use the "|" separator to specify the different node types that could be assigned to N_1 and N_2. The correspondences between the node types in a user query and those in the query graph are implicitly shown.

There are three kinds of basic node relations in user queries: N_1 calls N_2, N_1 controls N_2, and N_1 dataDepends N_2. We describe the constraints on the node types for each kind of node relations expressed in user queries and how the relations are converted into query graphs as follows:

1) N_1 **calls** N_2**.** For this kind of node relations, both N_1 and N_2 must be of type *function* in DQL^T for a query expression to be valid. This query is converted into a query graph containing two nodes as shown in Figure 3(a).
2) N_1 **controls** N_2**.** For this kind of node relations, N_1 must be of type *controlPoint*, while N_2 can be of any one of the types: *function*, *assignment*, *controlPoint*, or *statement*. This query is converted into a query graph containing two nodes as shown in Figure 3(b).
3) N_1 **dataDepends** N_2**.** For this kind of node relations, N_1 and N_2 can be of any type except *declaration*. Figure 3(c) shows how the query is converted into a query graph when N_1 is not a function. Figure 3(d) shows the query graph formed when N_2 is a function.

B. Graph Matching

After the above step is done, we have a small query graph and a large enriched SDG. We want to match the small graph against the big graph.

This task is computationally expensive, especially when we want to enable transitive (in addition to *oneStep*) dependence queries. Given a query graph consisting of n nodes, a naive approach would be to enumerate all possible n-node subgraphs of the enriched SDG and check if the constraints specified in the query is satisfied by each of the subgraph. This would incur a very high cost which is not practical for our purpose.

To tackle this challenging graph matching problem, we perform some heuristics to exclude nodes and edges that could not be part of a search result, and break down the transitive dependence queries into a number of graph reachability queries so that the state-of-the-art technique in graph reachability can be utilize to process these queries.

Our graph matching algorithm is mainly comprised of three components, as shown in Figure 2. Its high level description is as follows. We first identify the nodes in the enriched SDG that potentially match each node in the query graph. We refer to the set of potentially matching nodes as *potential node set (PNS)*, and denote the PNS for a node n in the query graph as PNS_n. We then identify the set of edges that connect nodes in PNSs and potentially satisfy the constraints specified in the query graph. We refer to the set of the edges as *potential edge set (PES)*. We keep these PNSs and PESs in a data structure we refer to as a *work list*. We further *filter* away unpromising nodes and edges in PNSs and PESs that could not be part of a search result. Finally, we *compose* answer sets by considering all combinations of the nodes and edges in PNSs and PESs.

More detailed descriptions are given below:

Work List Construction. We take in a query graph and topic-enriched SDGs and convert them into a work list. This is done by converting each node n and edge e in the query graph to the corresponding PNS_n and PES_e.

PNS Construction. Each node in a DQL^T query could be specified with its type, textual description, syntactic restriction, and semantic topics, as defined by $tlist$, $textOp$, $ucond$, $cond$, etc. in Table I. These constraints are called *unit conditions* and *topic conditions* of a query. The PNS_n for a node n is obtained by getting the nodes satisfying these conditions

PES Construction. We construct one PES for every edge e in the query graph, denoted PES_e. Each PES_e is the set of edges in SDG that connect two nodes in two different PNSs whose corresponding nodes are connected in the query graph. To check if two nodes are connected we use a graph reachability indexing algorithm [7].

Node & Edge Filtering. In this step, we filter away nodes and edges that are not possible to be part of a search result (called *unpromising*) based on the intuition that all matching nodes and edges in the SDG should have the same connectivity as the nodes and edges in the query graph. The purpose of the filter is to remove all unpromising nodes and edges, and reduce unnecessary checks when composing PNSs and PESs into final search results. Due to space limitation, we leave the description of the filtering process to [13].

Answer Set Composition. Each node in the PNSs after the filtering step is part of one or more search answers. There are potentially many answers to a query based on the sizes of the PNSs and the PESs. In this last step, we compose nodes and edges in PNSs and PESs to form search answers. The main idea to take every possible combination of the nodes from the PNSs and see whether each of the combination including associated edges from the PESs can match the edges in the query graph. Finally, we translate each valid combination back to code snippets as a search result where the line numbers and the corresponding source codes are shown.

Dataset	Approach	TP	TQ	TRF	TT
Expat_v32	[14]	1.7	64.7	2.1	68.5
	ours	0.3	6.0	0	57.3
Expat_v38	[14]	0.9	30.4	1.1	31.5
	ours	0.8	0.5	0	1.3
GPSbabel_v1071	[14]	3.3	640.2	3.6	647.1
	ours	0.6	1.0	0	13.2
GPSbabel_v1200	[14]	3.3	265.2	0.9	289.4
	ours	0.7	0.1	0	0.81

TABLE II
PERFORMANCE COMPARISON (IN SECONDS)

VI. EMPIRICAL EVALUATION

In this section we first describe our scalability evaluation. We then present four case studies to show the utility of our topic-enriched dependence search tool on a number of software engineering tasks on various systems.

Scalability Evaluation. We compare the speed of our processing engine with the previous engine proposed by Wang *et al.* [14]. We use the dataset studied in their paper for comparison. The results of the evaluation are shown in Table II. Column "TP" represents the time consumed for pruning unpromising nodes and edges before actual search. Column "TQ" represents the time for performing queries. In the case of [14], the column also includes the time spent in building graph indices for each query, while our approach performs one-time indexing and just needs to load index when required by a query. Column "TRF" represents the time of post-filtering on search results which is not applicable to our approach. The last column "TT" is the total time for the whole query process, including all necessary graph indexing costs. From Table II, we notice that on average our approach is more than 130 times faster.

Case Studies. We next describe some case studies showing the utility of our code search engine.

Task One. The first searching task is from the Expat project version 2002-05-17 which is a C language project. The task was also used in [14]. One comment of the code change mentioned failed `MALLOC()` calls that can cause potential memory leaks. The code before the change is shown below:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf)
    return XML_ERROR_NO_MEMORY;
```

The code after the change is shown as below, with an added call to `free`:

```
tag = MALLOC(sizeof(TAG));
if (!tag)
    return XML_ERROR_NO_MEMORY;
tag->buf = MALLOC(INIT_TAG_BUF_SIZE);
if (!tag->buf){
    free(tag);
    return XML_ERROR_NO_MEMORY;
}
```

Our task is to find all similar code that may return without freeing certain memories. A sample query for this purpose in our DQL^T syntax is as follows:

Topics: *malloc*;

Nodes: *func A, func B, var C, var D, ctrlPoint E ofControlType if, stmt F return*;

Relations: *C dataDepends A, D dataDepends B, D isFieldOf C, E dataDepends D, E oneStep controls F*;

Targets: *E*

In the query, the topic is $malloc$. A corresponds to a function call to `MALLOC` whose return value is used to initialize a variable C; another variable D, a field of C, is assigned the value from another call and used in a control point E, and one of E's branch may call `return`.

In this task, the developers actually changed two places in the program. Our approach finds both of the places, plus four other places, which are false positives, while Wang's approach [14] finds the two target places but brings 36 false positive. Our approach provides better precision.

Among the four false positives (*i.e.*, the four places not changed by the developers), one actually missed a call to `free` too; the developers missed the place when fixing the memory leak bugs. This also suggests our code search engine can be useful for detecting inconsistent changes to help reduce bugs.

Task Two. The second task is from *Media Player Classic–Home Cinema* version 1639. The comment of the code change is to fix the memory leak in decoder initialization functions. The aim is to find all the memory leaks which are caused by `returns` before freeing relevant memory storing structure variables. An example is as below.

```
Decoder_amrState * s; ...
if ( ( s->lsfState = ( D_plsfState * )
        malloc( sizeof( D_plsfState ) ) ) == NULL ) {
      fprintf( stderr, ... );
      return -1;
}
```

The code may cause memory leak since it does not free the memory of occupied by the variable `s` and its pointer fields other than `lsfState`. In this code, the assignment `s->lsfState = ...malloc...` has one-step data dependency on `s` and the return value of `malloc`. The control point (`if (...== NULL)`) has one-step data dependency on the assignment, and controls `fprintf` and the `return` statement. The topics for the changes codes are $decoder$ and $init$ according to the change comment. Therefore, our user query is as below.

Topics: *decoder, init*;

Nodes: *var A, func B contains "malloc", ctrlPoint C ofControlType if, stmt D return, assgn E, var X*;

Relations: *X isFieldOf A, X oneStep dataDepends E, E oneStep dataDepends B, C oneStep dataDepends X, C oneStep controls D*;

Targets: *C*

In this query, $isFieldOf$ is again used to relate a field to its containing structure. C is the $target$ because the developer wants to locate only the branches that may need calls to `free`.

In this task, there are 10 targets in the code, and our approach finds all the targets with one false positive. If no topic is used for search, the search based on dependencies only finds all the targets, but brings 14 false positives. If the search is given a constraint that the name of the involved functions should contain both `decoder` and `init` without using the topics, one target is missed, since this target is in a function `Post_Filter_init` whose name does not contain "decoder". Search with topics can find this target since this function has latent semantic relationship with "decoder" even though its name contains no "decoder".

VII. Conclusion

We propose a code search approach that utilizes both of the topics and the dependence relations in a code base for more accurate results. Our approach provides a query language that allows users to specify both free-form topics and complex dependence relations. We design and implement algorithms that can utilize topic modeling techniques to refine dependence graph queries. As a result, our approach can not only return more relevant search answers, but also is more scalable for processing transitive dependence queries, achieving on average an order of magnitude speedup than a previous state-of-the-art approach. We have evaluated our tool on several software maintenance tasks in various software systems and demonstrated the utility of our code search engine.

Acknowledgement. We would like to thank the authors of [7], [11] who have kindly provided the implementations of their algorithms to us. This work is partially funded through the LEE Foundation Fellowship (C220/T050034) from the Office of Research, Singapore Management University.

References

[1] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE*, 2008, pp. 321–330.

[2] GrammaTech, *CodeSurfer*, http://www.grammatech.com/.

[3] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. M. Cumby, "A search engine for finding highly relevant applications," in *ICSE*, 2010, pp. 475–484.

[4] J. Henrard and J.-L. Hainaut, "Data dependency elicitation in database reverse engineering," in *CSMR*, 2001, pp. 11–19.

[5] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *ICSE*, 2009, pp. 232–242.

[6] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones." in *ICSE*, 2007.

[7] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *SIGMOD*, 2010.

[8] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, 2009.

[9] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *WCRE*, 2004, pp. 214–223.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage." in *ICSE*, 2011.

[11] X.-H. Phan and C.-T. Nguyen, *JGibbLDA*, http://jgibblda.sourceforge.net/.

[12] S. Thummalapenta and T. Xie, "ParseWeb: A programmer assistant for reusing open source code on the web." in *ASE*, 2007.

[13] S. Wang, D. Lo, and L. Jiang, "Code search via topic-enriched dependence graph matching," School of Information Systems, Singapore Management Univ., Tech. Rep., 2011.

[14] X. Wang, D. Lo, J. Chang, L. Zhang, H. Mei, and J. Yu, "Matching dependence-related queries in the system dependence graph," in *ASE*, 2010.

[15] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns." in *ECOOP*, 2009.