Inferring Semantically Related Words from Software Context

Jinqiu Yang and Lin Tan
University of Waterloo, Waterloo, ON, Canada
{*j223yang, lintan*}*@uwaterloo.ca*

***Abstract*—Code search is an integral part of software development and program comprehension. The difficulty of code search lies in the inability to guess the exact words used in the code. Therefore, it is crucial for keyword-based code search to expand queries with semantically related words, e.g., synonyms and abbreviations, to increase the search effectiveness. However, it is limited to rely on resources such as English dictionaries and WordNet to obtain semantically related words in software, because many words that are semantically related in software are not semantically related in English. This paper proposes a simple and general technique to automatically infer semantically related words in software by leveraging the context of words in comments and code. We achieve a reasonable accuracy in seven large and popular code bases written in C and Java. Our further evaluation against the state of art shows that our technique can achieve a higher precision and recall.**

***Keywords*-Semantically related words; code search; program comprehension**

I. Introduction

Code search is an integral part of software development; developers spend up to 19% of their development time on code search [26]. It becomes more difficult for one developer to understand and remember every piece of a software project, as software becomes larger and more complex, software is typically developed by hundreds of or thousands of programmers across decades, and developers frequently join and depart from the software development process. In order to find relevant code segments, code search is becoming a crucial part of software development and program comprehension.

The search for relevant code segments is difficult, because there is a small chance (10-15%) that developers guess the exact words used in the code [16]. For example, if developers want to find methods that disable interrupts in the Linux kernel, a simple regular expression based search "*disable*interrupt*"[1] will miss the functions "`disable_irq`" and "`mask_irq`". Both functions disable interrupts. The problem is the mismatches between the words *interrupt* and *irq* and between the words *disable* and *mask*. Similarly, if we search for "*add*auction*" in jBidWatcher, the method "`AuctionsManager.newAuctionEntry(String)`" will not be returned, although it is related to adding an auction entry.

[1]It is possible to perform a relaxed search to find method names that contain either the word *disable* or the word *interrupt*, but such an approach generally retrieves too many irrelevant matches to be useful.

Researchers proposed to expand search queries with semantically related words (e.g., synonyms and abbreviations) for more effective searches [38]. However, leveraging an English dictionary [9] and WordNet [11] for obtaining semantically related words is limited in the software domain, because many words that are semantically related in *software* are not semantically related in *English*. In the previous example, the words *disable* and *mask* are not related words either in an English dictionary [9] or WordNet [11]. Similarly, *interrupt* and *irq* are not semantically related in the English dictionary or WordNet. A recent study evaluated six well known techniques for discovering semantically related words in English and showed that these techniques are limited in identifying semantically related words in software [40]. The best technique needs to find over 3,000 pairs of words in order to discover 30 out of the 60 semantically related word pairs in the gold set.

If we can automatically discover semantically related words from software, it would not only improve search tasks, but also benefit other software engineering tasks. For example, aComment [43] leverages semantically related words to find comments that have similar meanings in order to check these comments against source code to detect bugs. Currently, aComment requires its users to manually specify synonyms and paraphrases, which is challenging since it requires the users to have domain knowledge about the target software. In addition, the ad hoc process is likely to miss important synonyms and paraphrases. An automated approach can potentially discover more synonyms and paraphrases and reduce the manual effort required.

Therefore, we propose to automatically identify semantically related words by leveraging the *context* of words in comments and code. This includes relations such as synonyms, antonyms, abbreviations, related words, etc., all of which are useful for code search. We use *semantically related word pairs* or the shorter *rPairs* to denote a pair of semantically related words and phrases. Our intuition is that if two words or phrases are used in the same context in comment sentences or identifier names, then they likely have syntactic and semantic relevance. For example, by examining the two comment sentences from the Linux kernel—"`Disable all interrupt sources.`" and "`Disable all irq sources.`", we can learn that the words *interrupt* and *irq* are likely to be related because both words appear in the same context. In this particular case,

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

the two words have the same meaning. Similarly, from two functions—"`void mask_all_interrupts()`" and "`void disable_all_interrupts(...)`"—we can infer that the word *mask* and the word *disable* form an rPair in this context. In addition to learning nouns and verbs that have similar meanings, we can learn adjectives with similar meanings. For example, we can infer that the two adjectives *disabled* and *off* have the same meaning from the two comments—"`Must be called with interrupts disabled.`" and "`It MUST be called with interrupts off.`"

Shepherd et al. [38], [39] extract verb-DO (Direct Object) pairs from software which can be leveraged to identify semantically related words. For example, if they discover verb-DO pairs *(add | element)* and *(find | element)* in iReport, they would suggest the word *find* to users to expand their query "add element" in iReport [38], because (*add*, *find*) are considered semantically related. This paper differs from the previous work mainly in the following aspects. First, the previous work replies on heuristics regarding the naming convention and the structure of code identifiers and comments. For example, they use different heuristics to extract the DO from a method name, depending on whether a verb exists in the method name, where the verb is, and what the verb is. Such heuristics are manually designed by the authors and may not generalize if the naming convention or structure is not followed. Our technique *requires no heuristics about the naming convention or the structure of the code identifiers and comments*[2], and can potentially be applied to a broader spectrum of code bases.

Second, the previous work focuses on verb-noun (verb-DO) relations; therefore may miss the opportunity to discover rPairs from other relations such as noun-noun relations. For example, the two comments "`Min of spare threads`" and "`Min of spare daemons`" contain no verbs, so that the previous technique cannot discover the semantically related word pairs *(thread, daemon)* from them, while our technique can. In addition, the previous work is likely to miss other types of rPairs such as adjective-adjective rPairs.

Third, the previous technique leverages Natural Language Processing (NLP) techniques, such as part-of-speech (POS) tagging and chunking, which are trained from general English text such as the Wall Street Journal, not from software. When applied to the software domain, these techniques can cause inaccuracies in rPair extraction. For example, it would fail to identify the verb-DO pair from "`newParameter()`", as *new* is a noun in English. But in the software context, *new* is commonly used as a verb to refer to creating memory for a new object. This inaccuracy prevents the previous techniques from discovering the rPair *(new, add)* that our technique can discover, because our technique ignores the part of speech. The detailed comparison is discussed in Section V.

[2]Except that we break method names into words based on camel case and underscore, which is also used by the previous work

This paper makes the following contributions.

- We propose a context-based approach to automatically infer semantically related words by leveraging the *context* of words in comments and code. Our technique can be used as a building block for many other software engineering tasks including code search [20], [38] and software bug detection [43].
- The proposed technique identifies semantically related words with a reasonable accuracy in seven large and popular code bases written in C and Java—the Linux kernel, Apache HTTPD Server, Apache Commons Collections, iReport, jBidWatcher, javaHMO, and jajuk. We classify the semantically related word pairs into five categories—*synonym*, *related*, *antonym*, *near antonym*, and *identifier*. The majority of the identified semantically related word pairs cannot be found in WordNet [11] or an English dictionary [9].
- Our further evaluation against the state of art [18], [38] shows that our overall recall and precision in discovering semantically related word pairs is higher. Since automatically expanding queries with inappropriate synonyms may produce worse results than not expanding [40], it may be beneficial to leverage techniques similar to previous work [20], [38] to allow developers to pick from a list of semantically related words. Since our technique has *higher recall* (finds more rPairs) with *higher precision* (more of the pairs discovered are truly rPairs), it can help developers find more relevant code segments and comments, as well as find them more quickly because developers will examine fewer incorrect rPairs.

Although our technique presents new opportunities to discover more semantically related words and improves the performance of discovering them, the absolute precision is relatively low due to the inherent difficulty of the task. Therefore, we discuss techniques that can potentially further improve the precision in Section IV-A.

II. Design

Our goal is to automatically learn semantically related words and phrases by leveraging the context of words and phrases in comments and code. Examples in Table I help illustrate how semantically related words and phrases can be learned from comments and code. Column 'Context Type' shows whether the context is from comments or source code: comment-comment indicates that both contexts are from comments; code-code means that both contexts are from source code; and comment-code denotes that one context is from comments, and the other context is from source code. For example, both of the two jajuk comments "`None mounted file for this track.`" and "`None accessible file for this track.`" state that a file associated with the track is missing. Since the words *mounted* and *accessible* are surrounded by the same context, "`None`

Table I
LEARNING SEMANTICALLY RELATED WORD PAIRS FROM CONTEXT. THE COMMENT AND CODE EXAMPLES ARE REAL COMMENTS AND CODE SEGMENTS FROM THE SEVEN CODE BASES USED IN OUR EVALUATION.

Context	Semantically Related Word Pairs	Context Type
`Must be called with interrupts` **`disabled`**`.` `It MUST be called with interrupts` **`off`**`.`	(disabled, off)	Comment-Comment
`Disable all` **`interrupt`** `sources.` `Disable all` **`irq`** `sources.`	(interrupt, irq)	Comment-Comment
`Always` **`called`** `with interrupts disabled.` `Always` **`invoked`** `with interrupts disabled.`	(call, invoke)	Comment-Comment
`None` **`mounted`** `file for this track.` `None` **`accessible`** `file for this track.`	(mounted, accessible)	Comment-Comment
`Serializes` `this map` **`to`** `the given stream` **`Deserializes`** `this map` **`from`** `the given stream`	(serialize, deserialize) & (to, from)	Comment-Comment
`Min of spare` **`threads`** `Min of spare` **`daemons`**	(thread,daemon)	Comment-Comment
`Empty map with the specified maximum` **`size`** `Empty map with the specified maximum` **`capacity`**	(size,capacity)	Comment-Comment
`Gets the value` **`associated`** `with the key` `Gets the value` **`mapped`** `with the key specified`	(associate, map)	Comment-Comment
`get a node's` **`parent`** `get a node's` **`left child`**	(parent, left child)	Comment-Comment
`An iovec to store the` **`headers`** `sent` **`before`** `the file` `An iovec to store the` **`trailers`** `sent` **`after`** `the file`	(header, trailer) & (before, after)	Comment-Comment
`it was finally rewritten to a` **`remote URL`** `it was finally rewritten to a` **`local path`**	(remote URL, local path)	Comment-Comment
`mask``_all_interrupts()` **`disable`**`_all_interrupts(...)`	(mask, disable)	Code-Code
`add``Parameter(...)` **`new`**`Parameter()`	(add, new)	Code-Code
`File``TypeFileFilter()` **`Directory`**`TypeFileFilter()`	(file, directory)	Code-Code
`Initialize` `signal names` **`setup`**`_signal_names(...)`	(initialize, setup)	Comment-Code
`Alloc` `a net device` **`add`**`_net_device(...)`	(alloc, add)	Comment-Code

`... file for this track.`", we consider the word pair (*mounted*, *accessible*) an rPair.

A. An Overview of the Analysis Process

Our analysis technique takes a code base and a stopword list as input, and outputs semantically related word pairs. The analysis process consists of four steps: (1) *parsing comments and code*: given a code base, we first parse it to extract all the comment sentences and method names, and convert each of them into a sequence of words; (2) *clustering comments and code*: we cluster the word sequences based on whether they contain at least one common word to reduce the overhead of pairwise comparison in the next step, which is a critical step for our technique to scale up to large code bases such as the Linux kernel; (3) *extracting semantically related word pairs*: we calculate the similarity between a pair of word sequences and extract the corresponding rPairs if the context is similar; and (4) *refining semantically related word pairs*: we finally refine the rPairs by using stemming to remove pairs with the same roots, merging duplicate word pairs, normalizing words, and generating transitive rPairs.

B. Parsing Comments and Code

We extract all comment blocks from source code files and use a sentence segmentator to split them into comment sentences. Each comment sentence is broken down into a sequence of words by using space as the delimiter. For example, the comment sentence "`Called with interrupts disabled`" is represented as a sequence consisting of four words (case insensitive): <`called`, `with`, `interrupts`, `disabled`>. Similarly, we extract method names from source code files, and split them into words based on camel case or underscore. To minimize the dependency on naming convention and code structure related heuristics, our analysis ignores return types and parameters.

A sentence segmentator for English sentences does not work well for code comments mainly because incorrect punctuation is common in comments. Therefore, in addition to regular sentence delimiters, i.e., "!", "?", and ";", we use "." and spaces together as sentence delimiters instead of using "." alone, and consider an empty line and the end of a comment as the end of a sentence [43].

In order to discover semantically related identifiers and avoid duplicate analysis, we do not break identifiers in comments into multiple words based on camel case or underscore. For example, we can learn that the `apr_pool_clear` and `apr_pool_destroy` are semantically related methods in HTTPD from comments "`If you do`

`not have apr_pool_clear in a wrapper`" and "`If you do not have apr_pool_destroy in a wrapper`".

C. Clustering Comments and Code

It is expensive to conduct pairwise comparison for a large number of sequences. For example, the Linux kernel contains 519,168 unique comment sentences. Pairwise comparison requires us to compare on the order of *100 billion* (134,767,706,112) pairs of word sequences to check if we can find rPairs from them. This is already the number after we filter out sequences that are too short or too long as described later in Section II-D. We ran the experiment on an Intel Core 2 Duo 3.06 HZ machine, and the pairwise comparison does not finish in one day.

To speed up the process, we want to reduce the number of pairwise comparisons. Our intuition is that there is no need to compare two sentences that do not share a single word. Therefore, we group sequences into *clusters*, one cluster for each word, where each cluster contains all the sequences that contain the word. We do not build clusters for words in the stopword list, which are words that appear frequently in English and software such as 'a', 'an', 'the', 'that', 'this', etc. Sharing only these non-essential words do not increase the similarity of the context for discovering rPairs. We then conduct pairwise comparisons within each cluster. Since each cluster contains much fewer number of word sequences, this approach can significantly reduce the number of pairwise comparisons. For example, this step speeds up the analysis process for the Linux kernel by almost 100 times: all the comments are divided into 123,404 clusters, and the total number of pairwise comparisons has been reduced to 90,483,147, which translates to only one hour on the same machine.

D. Extracting Semantically Related Word Pairs

The main step of the extraction process is to calculate the similarity between two word sequences and extract the corresponding word pairs if the similarity is higher than a given *threshold*. Since sequences are not always lined up from the first word, e.g., <`must`, `be`, `called`, `with`, `interrupts`, `disabled`> and <`it`, `must`, `be`, `called`, `with`, `interrupts`, `off`>, we apply the Longest Common Subsequence (LCS) algorithm to find the longest overlapping subsequences (not necessarily continuous) between two sequences.

We define the similarity measure as

$$SimilarityMeasure = \frac{\text{Number of Common Words in the Two Sequences}}{\text{Total Number of Words in the Shorter Sequence}}$$

If the similarity measure of a pair of word sequences is greater than or equal to the *threshold* (whose default value is *0.7* for the comment-comment context) and not 1 (meaning that the two sequences are identical), we extract rPairs from the differences between the two subsequences.

Our technique can find semantically related phrases, not only semantically related words. For example, from the sequences <`get`, `a`, `nodes's`, `parent` > and <`get`, `a`, `nodes's`, `left`, `child` >, we find that the longest common subsequence is <`get`, `a`, `nodes's` >, and that phrases/words (*parent*, *left child*) are semantically related, because the $SimilarityMeasure$ is 0.75, which is greater than the default threshold. The rPair (*remote URL*, *local path*) is another phrase discovery example (Table I).

In addition, we can find more than one rPair from two sequences. For example, from the sequences <`an`, `iovec`, `to`, `store`, `the`, `headers`, `sent`, `before`, `the`, `file`> and <`an`, `iovec`, `to`, `store`, `the`, `trailer`, `sent`, `after`, `the`, `file`>, we can infer two rPairs (*header*, *trailer*) and (*before*, *after*).

In addition to the *threshold*, three additional parameters are used to control the rPair extraction process: *shortest*, *longest*, and *gap*. Our technique only analyzes word sequences whose length is greater than or equal to *shortest* and less than or equal to *longest*, where sequence length is defined as the number of words in a sequence. Our technique only performs pairwise comparisons between two sequences whose length difference is *gap* or less.

E. Refining Semantically Related Word Pairs

We finally refine the detected rPairs. We remove rPairs that contain words in the stopword list, e.g., (*a*, *the*); and we use stemming to remove word pairs with the same roots, e.g., (*call*, *called*). In addition, since the same rPairs may be discovered from multiple pairs of sequences, we merge the word pairs as one rPair. For example, we can learn that (*interrupt*, *irq*) is an rPair from the two relevant comments in Table I, as well as the two sequences <`were`, `called`, `from`, `interrupt`, `handlers`> and <`called`, `from`, `irq`, `handlers`>. We consider it as one rPair only, and increase the support for this rPair, which can be leveraged to rank the rPairs.

Lastly, we normalize words to their base forms. For example, we normalize the rPair (*called, invoked*) to (*call*, *invoke*), and normalize the rPair (*threads*, *daemons*) to (*thread*, *daemon*). Stemming is inappropriate for this normalization step, because a stemmer will revert words to their stems (e.g., *invoked* to *invok*), most of which are not words. We build a reversely mapped dictionary that can return the base form of a word, given the derived form (e.g., past participles and plural nouns) of the word. We build the reversely mapped dictionary from an English dictionary. We normalize an rPair only if both words can be normalized. For example, the rPair (*disabled, off*) is not normalized to (*disable, off*) because the word *off* is already in its base form.

We introduce *transitive rPairs*. If (W1, W2) and (W1, W3) are rPairs, (W2, W3) is a transitive rPair that requires one transition. If (W2, W4) is also an rPair, then (W3, W4) is a transitive rPair after two transitions. Considering transitive rPairs increases recall but reduces precision; our evaluation uses no transitive rPairs unless stated otherwise.

Table II
EVALUATED SOFTWARE. LOCOMMENT IS LINES OF COMMENTS. * VERSIONS OF IREPORT, JBIDWATCHER, JAVAHMO, JAJUK ARE THE SAME AS [38]

Software	Short Name	Version	Source	Description	LOC	LOComment	Language
The Linux kernel	Linux	3.3	[10]	Operating System	9,823,623	2,135,655	C
Apache HTTPD Server	HTTPD	2.2.21	[3]	Web Server	231,526	70,229	C
Apache Commons Collections	Collections	3.2.1	[2]	Libraries and Utilities	55,398	40,994	Java
iReport	iReport	1.2.2*	[5]	Report Generator	74,506	18,614	Java
jBidWatcher	jBidWatcher	1.0pre6*	[8]	eBay Auction Monitor	23,052	5,596	Java
javaHMO	javaHMO	2.4*	[7]	Media Server	25,988	7,784	Java
jajuk	jajuk	1.2*	[6]	Music Player	30,679	13,545	Java

III. EVALUATION METHODS

We evaluate our technique on seven open source projects (Table II). Because method names are typically much shorter than comment sentences, we use different parameters for the comment-comment, code-code, and comment-code comparisons. For comment-comment comparisons, the parameter configuration is *shortest*=4, *longest*=10, *gap*=3, and *threshold*=0.7; for code-code comparisons, the parameter configuration is *shortest*=2, *longest*=4, *gap*=0, and *threshold*=0.5; and for comment-code comparisons, the parameter configuration is *shortest*=2, *longest*=6, *gap*=1, and *threshold*=0.6.

We perform three sets of evaluation experiments.

A. Experiment: rPair Extraction Accuracy and Comparison with WordNet and a Dictionary

We randomly sample 300 rPairs from all the rPairs generated for each project—100 rPairs extracted from the comment-comment context, 100 from the code-code context, and 100 from the comment-code context. We then manually read these rPairs and the corresponding word sequences to verify if the rPairs are correct rPairs. If fewer than 100 rPairs are extracted from one type of context in a code base, we manually verify all of the rPairs learned from that context in that code base. The *accuracy* is measured as the number of correct rPairs in a sample over the total number of rPairs in the sample. We further classify the correct rPairs into five categories—*synonym*, *related*, *antonym*, *near antonym*, or *identifier*, whose definition and examples are shown in Section IV-A. To reduce subjectivity, two people verify these results. In addition, we check how many rPairs cannot be found in WordNet [11] or a dictionary [9].

B. Experiment: Search-Related Evaluation

Previous work [38] builds a code search tool that expands search queries with alternative words learned from verb-DO pairs. For example, when developers search for "add textfield" in iReport, the tool will suggest words including *element*, *keyword*, and *token* for developers to select from to expand the initial query to queries such as *add element*, *add keyword*, *add token*, etc. These words are objects (DOs) that appear together with the verb *add* in iReport. To evaluate the technique, they manually identify the methods related to the concern "add textfield", referred to as *method gold set*, and check if such query expansions can improve the search effectiveness.

Since our technique is only a building block for search tools, we compare the precision and recall of our rPair extraction results with the rPair extraction results of the previous work [38], [39], on the *rPair gold set* inferred from the same search tasks [1] used by the previous work. For example, their method gold set for the search task "add auction" in jBidWatcher includes methods "`AuctionsManager.newAuctionEntry(String)`" and "`AuctionServer.registerAuction(AuctionEntry)`", meaning that when developers search for "add auction", these two methods should be matched. A keyword-based search for "add auction" in source code files will not find these methods. To locate them, we need to expand the query to "new auction" and "register auction". Therefore, we add two rPairs, (*add, new*) and (*add, register*), to our rPair gold set for the query word *add*. Note that the rPairs are added based on the method gold set. For example, the pair (*add, insert*) is not in our rPair gold set, because according to the method gold set, we do not need the word *insert* to locate the methods related to "add auction" in jBidWatcher. Since only eight of the nine search tasks from the previous work [38] require query expansion, we generate the rPair gold set for the eight search tasks.

For a fair comparison, we tune the previous technique to achieve the best performance, i.e., the highest recall, since it is harder to guess the words used in code, than to cross off false positives. First, we compare against their *latest and improved version* [18] (denoted by v-DO). Since the improved version analyzes only code but not comments, we can only compare our code-code analysis against their approach. If we add our comment-comment and comment-code analysis, our approach could find more rPairs as discussed in Section IV-B and Section V. Second, we relax one restriction of the v-DO technique to help it find rPairs that it may miss otherwise. For example, for the query "*load movie*", the v-DO technique would suggest verbs that appear together with *movie*, which do not include *start*, because *start* does not appear together with *movie*. If the user decides to expand the query with the suggested words, the v-DO technique would suggest new words based on

Table III
RPAIR EXTRACTION RESULTS. DIC DENOTES MERRIAM-WEBSTER ENGLISH DICTIONARY AND THESAURUS [9]. THE MARGIN OF ERROR IS CALCULATED WITH 95% CONFIDENCE LEVEL.

Software	rPairs	Sample Size	Correct rPairs	Accuracy	*Synonym*	*Related*	*Antonym*	*Near Antonym*	*Identifier*	*#Not* in Dic or WordNet
Comment-Comment										
Linux	108,571	100	47	47.0±9.8%	1	20	1	2	23	36
HTTPD	1,428	100	47	47.0±9.5%	1	24	6	1	15	44
Collections	469	100	74	74.0±8.7%	0	58	4	3	9	72
iReport	878	100	84	84.0±9.2%	0	42	7	1	34	80
jBidWatcher	111	111	71	64.0%	0	47	8	12	4	63
javaHMO	144	100	56	56.0±5.4%	1	28	4	4	19	51
jajuk	203	100	69	69.0±7.0%	4	54	6	5	0	65
Code-Code										
Linux	606,432	100	25	25.0±9.8%	1	21	1	2	0	25
HTTPD	1,727	100	25	25.0±9.5%	1	18	3	3	0	24
Collections	3,162	100	41	41.0±9.7%	2	34	3	2	0	37
iReport	1,849	100	47	47.0±9.5%	1	44	1	1	0	47
jBidWatcher	1,428	100	42	42.0±9.5%	1	36	1	4	0	42
javaHMO	685	100	35	35.0±9.1%	0	33	1	0	1	35
jajuk	746	100	48	48.0±9.1%	0	43	3	2	0	47
Comment-Code										
Linux	10,633	100	25	25.0±9.8%	0	22	0	3	0	25
HTTPD	43	43	12	27.9%	1	11	0	0	0	12
Collections	5	5	0	0	0	0	0	0	0	0
iReport	4	4	4	100%	0	4	0	0	0	4
jBidWatcher	0	0	0	0	0	0	0	0	0	0
javaHMO	0	0	0	0	0	0	0	0	0	0
jajuk	6	6	4	66.7%	0	4	0	0	0	4

the new query. Therefore, whether *start* will eventually be suggested is uncertain. We relax this restriction so that the v-DO technique can find (*load*, *start*) as an rPair if *load* and *start* appear together with some DO, not necessarily *movie*.

Based on the rPair gold set, we measure the *recall* as the number of rPairs in the gold set that a technique can discover over the total number of rPairs in the gold set. The *precision* is the number of rPairs in the gold set that a technique can discover over the total number of rPairs discovered by the technique that contain the original query word in the gold set (e.g., *add* and *load* in the previous examples).

C. Experiment: Sensitivity Evaluation

To understand how the *threshold* affects the performance of the proposed technique, ideally we want to vary the threshold, regenerate rPairs, and measure the precision and recall on a random sample of the rPairs. However, as the rPairs generated will be different with different threshold values, this evaluation approach requires a significant amount of effort on manually verifying the rPairs in the random samples. Therefore, as an approximation, we use the same random samples from the rPairs generated with our default threshold values (referred to as *default samples*), and measure the *recall* as the portion of the correct rPairs in a default sample that can be identified by our technique with a new threshold. The *precision* is the number of correct rPairs in the default sample that our technique can discover over the total number of rPairs in the default sample that our technique can discover.

D. Threats to Validity and Limitations

The search gold set and rPair gold set (introduced in Section III) may favor a certain technique. To minimize this threat, we evaluate our technique on the same search gold set used by Shepherd et al. [38] as we compare against their technique. This is unlikely to favor our technique. In addition, two authors confirm the rPair gold set to avoid subjectivity.

If a code base contains no comments and the methods are poorly named, our technique may be less effective. However, given that modern software often contains a large amount of comments [42] and meaningful identifiers, our technique should be applicable to a large body of software. A large amount of commented code may affect the performance; in the future, we can exclude commented code from our analysis to address this issue.

Our current implementation cannot tell if an rPair is *synonym*, *related*, *antonym*, *near antonym*, or *identifier*. Although all categories are useful for code search, it would be beneficial to distinguish these categories. In the future, we may leverage etymology to classify rPairs into the categories automatically.

IV. EXPERIMENTAL RESULTS

A. rPair Extraction Results

Table III shows the overall rPair extraction results on the seven evaluated code bases from the three types of contexts: comment-comment, code-code, and comment-code.

Table IV
SEARCH-RELATED RESULTS. CTX IS OUR CONTEXT-BASE TECHNIQUE, CTX_T IS CTX WITH TRANSITIVE RPAIRS, AND V-DO DENOTES THE PREVIOUS WORK [18].

Search Task	Software	rPairs in Gold Set	#rPairs	Precision			Recall		
				CTX	v-DO	CTX_T	CTX	v-DO	CTX_T
"Add Textfield"	iReport	`add->new,drop`	2	3.7%	0	0.3%	50.0%	0	50.0%
		`TextField->ReportPanel`	1	0	0	0	0	0	0
"Compile Report"	iReport	`report->directory`	1	0	0	0.3%	0	0	100%
"Gather Music Files"	javaHMO	`gather->find`	1	0	0	1.1%	0	0	100%
		`file->directory`	1	3.3%	0.4%	0.6%	100%	100%	100%
"Load Movie Listings"	javaHMO	`listing->container`	1	0	0	0	0	0	0
		`load->start`	1	20.0%	5.3%	1.1%	100%	100%	100%
"Add Auction"	jBidWatcher	`add->register,do,new`	3	5.5%	2.3%	1.0%	100%	66.7%	100%
		`auction->entry`	1	1.8%	0.3%	0.4%	100%	100%	100%
"Save Auction"	jBidWatcher	`save->preserve,backup,do`	3	0	1.6%	0.3%	0	33.3%	33.3%
"Set Snipe Price"	jBidWatcher	`price->currency`	1	0	0	0/0	0	0	0/0
"Play Song"	jajuk	`song->playlist,file`	2	0	0	0/0	0	0	0/0
		`play->launch`	1	0	0	0/0	0	0	0/0

We show the margin of error with 95% confidence level except for comment-comment of jBidWatcher and comment-code of HTTPD, iReport, and jajuk, of which we verify all extracted rPairs. We can see that the accuracy of the comment-comment context is the highest (47.0–84.0%), which is expected because comment sentences are generally longer than method names, which provides longer context for learning correct rPairs. In contrast, we learn fewer rPairs from the comment-code context, due to the disparity between comments and method names. However, the comment-code context does help us learn meaningful correct rPairs such as (*initialize*, *setup*) and (*alloc, add*), whose contexts are shown in Table I.

Column 'Not in Dic or WordNet' shows the number of rPairs that cannot be found in either WordNet [11] or an English dictionary and thesaurus [9]. These words and phrases are semantically related in software, but are not semantically related in English. This is very valuable because it is almost impossible for developers to guess all the semantically related words used in a given piece of software. Our results show that 713 out of the 756 correct rPairs in the seven projects cannot be found in either WordNet [11] or an English dictionary [9].

The breakdown of the correct rPairs into five categories is also shown in the table. *Synonym* denotes words that have the same meanings in software (including abbreviations), e.g., (*call*, *invoke*) and (*interrupt*, *irq*). *Related* denotes words that are semantically related but not the same, e.g., (*size*, *capacity*) and (*file*, *directory*). *Antonym* denotes words that have opposite meanings, e.g., (*serialize*, *deserialize*) and (*before*, *after*). *Near Antonym* denotes words that have almost opposite meanings, e.g., (*header*, *trailer*). The full contexts of these rPair examples are shown in Table I. *Identifier* denotes words that are semantically related code identifiers, such as method names, variable names, etc. All five types of rPairs are useful for code search and other software engineering tasks.

False Positives. Despite the challenging nature of the task, our technique has a reasonable accuracy. However, there is much space to further improve the accuracy. One main cause of false positives is that the shared context contains many common English words. For example, we mistakenly consider (*match*, *literal*) semantically related, from comments "`we have a match`", and "`we have a literal`". Another reason is that our design favors recall over precision; the threshold and the support (the number of contexts from which the rPairs can be learned) are set low, and the gap allowed is high. Despite the false positives, our techniques is valuable, because it is much easier for developers to cross off false positives than to guess the possible semantically related words used in software.

To reduce false positives, we plan to rank rPairs according to the importance of the words in the shared context (e.g., tf-idf scores), the similarity measure, the support, etc. In addition, we can leverage NLP techniques to generate the semantic paths [31] to infer rPairs more precisely. At the cost of a lower recall, users can increase the threshold and the support and decrease the gap to improve the precision.

B. Search-Related Results

Table IV shows the search-related results. Column 'rPairs in gold set' shows all the rPairs in our *rPair gold set*, which can help expand the search queries to find the relevant methods. For example, for the search task "add auction" in jBidWatcher, we need to expand the query word *add* with *new*, *register*, and *do* to locate the relevant methods.

Column 'CTX' is our context-base technique, column 'CTX_T' is CTX with transitive rPairs, and v-DO denotes the previous work [18], [38]. As stated in Section III, we tune the v-DO technique to reach its best performance. Without the tuning, the v-DO technique would potentially miss three additional rPairs, (*load*, *start*), (*add*, *do*), and (*file*, *directory*).

Overall, our context-based approach (CTX) outperforms the v-DO approach. For three search tasks, both techniques have a zero recall. For four out of the five remaining tasks, our context-based approach has higher recall and precision or same recall with higher precision. For example, our technique can find the rPair (*add*, *new*) in iReport, but the v-DO approach will miss it because NLP tools trained from general English text will not consider *new* a verb as discussed in Introduction. One caveat is that using these semantically related words to expand queries may locate more irrelevant method names. However, recent techniques [20], [22] may be leveraged to restrict the search context and scope to address this issue.

For the rPairs that both techniques can find, our technique (CTX) has a higher precision (by a factor of 2.4–8.3). This is because we use similarity measures to filter out irrelevant pairs, and we do not consider return types or parameters to minimize the dependency on naming convention and code structure related heuristics. Although these design choices may make our technique discover fewer rPairs, they did not cause our technique to miss any rPairs in the search-related rPair gold set that the v-DO technique can find.

Although our technique (CTX) has a higher precision than the v-DO technique, the precision of both techniques is relatively low, because (1) this is an inherently challenging task, and (2) we only count the particular rPairs in our gold set as true positives, and consider other correct rPairs discovered by the techniques as false positives, which can be useful for other search queries nonetheless. We can use the techniques discussed in Section IV-A to improve the precision.

The only case that the v-DO approach has a higher recall than our approach is for the rPair (*save*, *do*). The v-DO approach breaks the method name `DoSave` into two verbs, and generate two verb-DOs by combing the method name with the parameter name. Our technique does not attempt to break one method name into two sequences, therefore misses this rPair. However, our technique with transitive pairs (CTX_T) can find this rPair.

In addition, Table IV shows that by considering transitive rPairs with at most two transitions allowed (CTX_T), we can find three additional rPairs in the gold set with lower precisions, including two rPairs that neither the v-DO technique nor our technique without transitive rPairs (CTX) can find—(*gather*, *find*) and (*report*, *directory*). If we analyze comments as well, then an additional rPair (*listing*, *container*) will be identified.

C. Sensitivity Results

To understand how a higher threshold affects the precision and recall on the comment-comment analysis, we first experiment with thresholds 0.8, and 0.9. We found that threshold 0.8 significantly reduces the number of identified rPairs (recalls are lower than 0.5 for all seven projects) with much higher precision (7.3–56.3% improvement) for all projects except for HTTPD (a small improvement of 0.6%), and threshold 0.9 finds zero rPairs in our default samples for iReport, HTTPD, and the Linux kernel, and one rPair for the rest 4 projects. Since we favor recall, we then evaluate a lower threshold 0.75, which, however, still gives us low recalls (less than 0.5 for all projects) with precisions lower than those with threshold 0.8. Since the number of words in a sentence is integers, thresholds between 0.7 and 0.75 are either equivalent to 0.7 or 0.75; therefore, there is no need to evaluate them. In summary, threshold 0.7 finds more rPairs with reasonable precisions; therefore, it was chosen as our default value. If higher precision is preferred and lower recall is acceptable, 0.8 is a good choice. We did not conduct the same experiment for the comment-code and code-code context because method names are generally short, e.g., two words; therefore, threshold tuning is not meaningful.

V. Related Work

A. Extracting Semantically Related Word Pairs

The closely related work [38], [39] infers semantically related words in software, and leverages them to build a search tool that outperforms two existing approaches [34]. We have already discussed the main differences between the previous work and our context-based approach in Introduction, so we only summarize them here and provide more examples. First, the previous work replies on manually created heuristics, which may not generalize to other types of software and software written in different programming languages (e.g., non-object-oriented software such as the Linux kernel and HTTPD). Our technique requires no such heuristics and is effective in extracting rPairs from both object-oriented software and non-object-oriented software. Second, the previous work focuses on verb-noun (verb-DO) relations.

Third, the previous techniques leverage NLP techniques, whose models are trained from general English text, not from software. When applied to the software domain, these models can make mistakes. What is worse, comments and code identifiers are generally incomplete and grammatically incorrect, which may worsen the analysis inaccuracy problem. For example, two sentences with the same structure are analyzed differently because *interrupts* is a English word, while *irqs* is not. OpenNLP [4] (the same tool used by the previous work [38], [39]) tags the comment sentence from the Linux kernel "`called with interrupts disabled`" as $called_{<Verb>}\ with_{<Determiner>}\ interrupts_{<Noun>}\ disabled_{<Verb>}$, and then chunks it as $called_{<VerbPhrase>}\ [with\ [interrupts]_{<NounPhrase>}\ ]_{<PrepositionalPhrase>}\ disabled_{<Unknown>}$. Since the chunker cannot tag the last word *disabled* with the proper Chunk/Phrase-level tag, words *called* and *interrupt* will be considered as a verb-DO incorrectly if *called* is considered active voice, or

no verb-DO will be found if *called* is considered passive voice (from the previous paper [39], it is unclear whether the heuristics treat *called* as active or passive voice; therefore, we discuss both for completeness). However, a slightly different comment from the Linux kernel "`called with irqs disabled`" will be tagged as $called_{<Verb>}$ $with_{<Determiner>}$ $irqs_{<Adjective>}$ $disabled_{<Adjective>}$, and be chunked as $called_{<VerbPhrase>}$ $[with$ $[irqs$ $disabled]_{<NounPhrase>}$ $]_{<PrepositionalPhrase>}$. The verb-DO will be (*called* | *irqs disabled*), or no verb-DO is inferred. From these verb-DOs, previous work may consider *interrupts* and *irqs disabled* semantically related words by mistake, or it infers no semantically related words. Our technique can correctly identify *interrupts* and *irqs* as an rPair. This example shows that our technique is robust despite incomplete and grammatically incorrect comments. In addition, NLP analysis adds time complexity.

Since our technique ignores the part of speech, our technique has higher recall than the previous techniques. In addition, since we use similarity measures and consider the full context instead of just the verbs and DOs, our technique can be more precise. For example, from two comments "`initialize the product price`" and "`initialize the user name`", we would not consider the phrases *product price* and *user name* rPairs, since the similarity is only 50%, while the previous technique obtains two verb-DOs that share the verb *initialize*, and considers the two phrases semantically related.

The authors improved the original v-DO technique by leveraging phrasal concept and more advanced heuristics [18], [22]. The latest implementation uses specialized techniques to address the OpenNLP-related issues. However, the improved technique does not analyze comments, missing the opportunity to detect more rPairs. On the other hand, their approach analyzes return types and parameters, which may find more rPairs. In addition, the restrictions regarding verb-verb match may help filter out false positives. However, our evaluation in Section IV-B shows that our technique has better overall precision and recall in discovering rPairs than their latest implementation [18].

Techniques and resources that discover semantically related words in English [9], [11], [14], [25], [30], [31], [35], [46] are limited in discovering semantically related words in *software* [40]. Other work splits multi-word identifiers and discovers abbreviations in software [19], [27], [28]. Our technique finds general semantically related word pairs including abbreviations, which is complementary to the previous work. In addition, while the previous work uses statistical analysis, heuristics, and English dictionaries, we leverage the context of words in comments and identifiers.

B. Code Search

Keyword-based code search techniques have been developed [18], [20], [22], [34], [38], [33]. Since our technique infers semantically related words from software, it can be leveraged by these search tools to expand queries to further improve the search effectiveness. In addition, since our technique provides not only semantically related words but also the context, our technique could be leveraged by contextual-based search techniques [20] to improve the code search accuracy. Alternatives to keyword-based search include structural search [23], [24], [37], [47]. A recent study [21] investigates how to effectively combine global and local code search techniques.

C. Analysis of Natural-Language Text for Software

Previous work analyzes natural-language artifacts such as bug reports [13], [17], [29], [32], [36], [41], [44], [45], comments [42], API documentation [48], and identifier names [12], [15] for purposes such as detecting duplicate bug reports, identifying the appropriate developers to fix bugs, improving structure-field names, etc. This paper analyzes comments and code to discover semantically related word pairs. Different from some of these studies [12], [15], [42] that use NLP techniques such as POS tagging, chunking, and semantic role labelling, this paper chooses not to use NLP techniques for simplicity, efficiency, and generality. On the other hand, it is conceivable to use NLP techniques to generate NLP-related context to infer semantically related words, which remains as our future work.

VI. Conclusions and Future Work

We design and evaluate a general technique to automatically discover semantically related words in software by leveraging the context of words in comments and code. The proposed technique identifies semantically related words with a reasonable accuracy in seven large and popular code bases written in C and Java. Our further evaluation against the state of art shows that our overall precision and recall in discovering semantically related word pairs is higher. The semantically related words and the relevant context can be used by code search techniques to improve program comprehension and programmer productivity; they can also benefit other software engineering tasks such as bug detection for better software reliability.

In the future, we plan to use our technique to build comprehensive and accurate databases of semantically related words in software. Furthermore, we plan to explore the following techniques to reduce false positives: (1) rank the rPairs based on the similarity measure, the support, the gap, the importance of the words in the shared context, etc.; and (2) use NLP techniques such as POS tagging, chunking, and semantic role labelling to generate NLP-related context to infer semantically related words. In addition, we can analyze user manuals and other software documents to learn semantically related words.

ACKNOWLEDGEMENTS

We thank Emily Hill for providing us the verb-DO pairs from their latest SWUM implementation and Chen Liu for help with verifying the word pairs. We thank the anonymous reviewers for their feedback. The work is partially supported by the National Science and Engineering Research Council of Canada and a Google gift grant.

REFERENCES

[1] Action-oriented concerns. http://www.eecis.udel.edu/~gibson/context/action_oriented_concerns.txt.

[2] Apache Commons Collections. http://commons.apache.org/collections/.

[3] Apache HTTPD Server. http://httpd.apache.org.

[4] Apache OpenNLP. http://incubator.apache.org/opennlp/.

[5] iReport. http://jasperforge.org/projects/ireport.

[6] jajuk. http://www.jajuk.info/index.php/Main_Page.

[7] javaHMO. http://www.javahmo.sourceforge.net/.

[8] jBidWatcher. http://www.jbidwatcher.com/.

[9] Merriam-Webster English Dictionary and Thesaurus. http://www.merriam-webster.com/.

[10] The Linux Kernel. http://www.kernel.org.

[11] WordNet. Princeton University. http://wordnet.princeton.edu.

[12] S. L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. ICPC, 2010.

[13] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE*, 2006.

[14] S. Banerjee and T. Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *IJCAI*, 2003.

[15] D. Binkley, M. Hearn, and D. Lawrie. Improving identifier informativeness using part of speech information. MSR, 2011.

[16] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30, 1987.

[17] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *MSR'10*.

[18] E. Hill. *Integrating natural language and program structure information to improve software search and exploration.* PhD thesis, 2010.

[19] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. MSR, 2008.

[20] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. ICSE, 2009.

[21] E. Hill, L. Pollock, and K. Vijay-Shanker. Investigating how to effectively combine static concern location techniques. SUITE, 2011.

[22] E. Hill, L. L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *ASE*, 2011.

[23] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. ICSE, 2005.

[24] D. Janzen and K. De Volder. Navigating and querying code without getting lost. AOSD, 2003.

[25] J. J. Jiang and D. W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *CoRR*, 1997.

[26] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE*, 2005.

[27] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *ICSM*, 2011.

[28] D. J. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *WCRE*, 2010.

[29] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? – An empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[30] D. Lin. An information-theoretic definition of similarity. ICML, 1998.

[31] D. Lin and P. Pantel. Discovery of inference rules for question-answering. *Nat. Lang. Eng.*, 7, 2001.

[32] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR*, 2009.

[33] D. Poshyvanyk, A. Marcus, and Y. Dong. Jiriss - an eclipse plug-in for source code exploration. In *ICPC*, 2006.

[34] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with google. In *ICSM*, 2006.

[35] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *IJCAI*, 1995.

[36] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, 2007.

[37] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. ESEC-FSE, 2007.

[38] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD*, 2007.

[39] D. Shepherd, L. Pollock, and K. Vijay-Shanker. Towards supporting on-demand virtual remodularization using program graphs. AOSD, 2006.

[40] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. ICPC, 2008.

[41] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, 2010.

[42] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. iComment: Bugs or bad comments? */. In *SOSP*, 2007.

[43] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt-related concurrency bugs. In *ICSE*, 2011.

[44] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE*, 2008.

[45] J. woo Park, M. woong Lee, J. Kim, S. won Hwang, and S. Kim. CosTriage: A Cost-Aware Triage Algorithm for Bug Reporting Systems. In *AAAI*, 2011.

[46] Z. Wu and M. Palmer. Verbs semantics and lexical selection. ACL, 1994.

[47] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4, 1995.

[48] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE*, 2009.