Automatically Mining Software-Based, Semantically-Similar Words from Comment-Code Mappings

Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
{mjhoward, sgupta, pollock, vijay}@cis.udel.edu

***Abstract*—Many software development and maintenance tools involve matching between natural language words in different software artifacts (e.g., traceability) or between queries submitted by a user and software artifacts (e.g., code search). Because different people likely created the queries and various artifacts, the effectiveness of these tools is often improved by expanding queries and adding related words to textual artifact representations. Synonyms are particularly useful to overcome the mismatch in vocabularies, as well as other word relations that indicate semantic similarity. However, experience shows that many words are semantically similar in computer science situations, but not in typical natural language documents. In this paper, we present an automatic technique to mine semantically similar words, particularly in the software context. We leverage the role of leading comments for methods and programmer conventions in writing them. Our evaluation of our mined related comment-code word mappings that do not already occur in WordNet are indeed viewed as computer science, semantically-similar word pairs in high proportions.**

I. Introduction

Many software development and maintenance tasks involve traceability and code search; however, most tools' effectiveness is hindered by inconsistent language use between users and developers. The mismatch between vocabularies often results from using multiple semantically-related terminologies for the same concept, which are then related only by synonymy or, in some cases, by hyponymy or hypernymy. For example, a developer looking for a routine that "inserts a new task" would quickly agree that the method `addTask(Task newTask)` would satisfy their inquiry.

Expanding queries or corpora with synonyms and other semantically related words supplements the "bag of words" approach to traceability and search by suggesting highly related terms in place of failed queries [1]. In fact, Shepherd et.al. [2] showed that expanding a query with WordNet [3] synonyms, in conjunction with a sophisticated search algorithm, can locate relevant code more effectively than traditional searching mechanisms.

This material is based upon work supported by the National Science Foundation under Grant No. CCF 0915803.

Previous work [4] led us to hypothesize that synonyms and other semantic relationships can be used to improve automatically generated recommendations for program exploration from a starting point. Existing approaches [5], [6] can take advantage of this information by incorporating a textual component in addition to structural information [4]. In addition, synonyms and hypernyms can be used to locate and rank code clones [7], a common refactoring task. Clones that have similar structure and identifiers indicate more tightly coupled clones. Thus, synonyms could be incorporated into a code fragment distance analysis, so in addition to comparing types and values for distance, the identifiers could be compared by using semantic similarity. Techniques to recommend relevant code examples or other artifacts from open source repositories [8], [9] could benefit from semantic similarity. Rather than just using the structure and words of code fragments, the artifacts could be augmented with topic words. Semantic similarity is also useful when analyzing maintenance requests, such as in maintenance request assignment [10], and tools for finding differences between versions of software [11] can also benefit by using synonymous relations between words—for example, when the change between two versions involves renaming a method to a synonymous word.

Thus, Sridhara et.al. [12] investigated whether six publicly available, well known semantic similarity techniques that perform well on English text [13]–[15] are directly applicable to the software domain. In general, these strategies use the hierarchical structure of WordNet to produce a score that indicates the similarity relation between words, with higher scores indicating higher similarity. Their study indicated that all six English text-based approaches perform poorly above 50% recall when applied to the software domain. 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. In general, the number of returned results can be 10 times greater than the number of desired results, and much more for higher levels of recall. While the use of semantic similarity information is intended to improve the performance of software tools, the large number of returned results could instead be detrimental. The qualitative study results also suggested that one promising way to customize semantic similarity techniques for software is to augment WordNet with relations specific to software, possibly by mining word relations from software, as some pairs were identified due to WordNet being augmented with some very common software word relations, such as (write, save).

Yang and Tan [16] developed a context-based approach to automatically infer semantically related words by leveraging the context of words in comments and code. The approach is based on the key insight 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. After extracting comments and method names from the code and clustering them based on at least one common word in their parsed word sequences, the similarity between a pair of word sequences is calculated, the longest common subsequences are extracted as semantically similar, and related word pairs are inferred from them. The evaluation shows that this approach can find related word pairs accurately when they share a common context.

In this paper, we present an approach to automatically mine word pairs that are semantically similar in the software domain, with the goal of automatically augmenting WordNet for use in software engineering tools. Our work differs from Yang and Tan [16] in that we seek to find word pairs that are commonly related over many applications, while their approach identifies word pairs related within a given context (such as the pair auction - entry in the project jBidWatcher). Specifically, we leverage the role of leading comments of methods and programmer conventions in writing them as well as method signatures to automatically mine semantically similar verbs. A leading comment sentence and the method name are normally both expected to state the computational intent of the method. Hence, this pair is expected to be the same or be different, but regardless, express the same action (i.e., be semantically similar). We focus on verbs as we believe that higher accuracy can be achieved for these words by leveraging programmer conventions in documenting code, and we believe that verbs are the words most likely mismatched in vocabularies among developers, with many synonyms for word choice among different readers and writers of code. We believe our work is complementary to Yang and Tan [16], and they could be combined for certain software engineering tools.

Our key insight in mining semantically-similar verbs is to map the main action verb from the leading comment of each method to the main action verb of its method signature. While the intuition that they should be semantically similar is straightforward, automatically identifying them is not trivial. While many method names begin with a verb, many do not fall in the first position. We developed a part-of-speech tagging technique to help automatically identify the main action verb in a method signature. In addition, identifying the main verb in the leading comment also presents challenges, beginning with ensuring that the comment is descriptive of the method's actions, and additional challenges exhibited in the next section.

To address these challenges, the contributions of this paper include:

- an automated analysis of mined comment-code action verb mappings to rank the most common semantically-similar verbs in the software domain,
- a rule-based algorithm to automatically identify the main action verb in a descriptive leading comment for a method,
- a heuristic to automatically classify a comment as a descriptive leading comment,
- a heuristic to identify the main action word from a method signature,
- an empirical evaluation of the overall semantically-similar pair-mining strategy and its component steps. The evaluation shows that mining accuracy increases directly with frequency and the component steps perform with moderately high accuracy.

II. Overview: Semantically-Similar Word Mining

Figure 1 presents the phases of our automatic technique for extracting word pairs that are semantically similar particularly in the software domain. Our hypothesis is that the action verb used in the descriptive leading comment for a method is semantically similar to the action verb used in the signature of the documented method. We define a *descriptive leading comment* to be a comment block placed before a method signature that provides the reader with the overall summary of a method's actions. That is, a descriptive comment describes the intent of the code succinctly.

There are four main components to our approach. The first step filters out leading comments that are non-descriptive. Then, for each descriptive leading comment, we extract the main action word from the descriptive leading comment and the main action word from the method's signature. The main action of a method is defined to be a term that describes the collective intent of the method body. The comment-code word pairs form the candidate semantically-similar word pairs. We then analyze and rank the comment-code pairs to automatically generate a list of semantically-similar word pairs.

III. Challenges Through Examples

In Table I, the first example containing a leading comment and its documented method signature motivates our key insight that synonyms can be automatically mined from mapping the main action verbs in leading comments to the main action verb in the documented method signature. In this case, the first word, 'searches', in the leading comment maps to the first word in the documented method signature, 'find', resulting in the mined semantically-similar word pair (searches, find). Because methods mostly correspond to actions or operations, programmers often start the method name with the main action word. And, many developers start the leading descriptive comment with a verb that describes the main intent of the method. In these cases, the automatic mining is straightforward once the non-descriptive comments have been filtered out.

TABLE I
EXAMPLE COMMENT-SIGNATURE SCENARIOS

Leading Comment - Method Signature	Action Pair	Automation Challenges

`/** Searches an attribute. */`
`XMLAttribute findAttribute(String fullName){ ... }`
	searches - find	Trivial to identify pair searches-find from comment and code respectively

`/** Cancels the current HTTP request. */`
`void jsxFunction_abort(){ ... }`
	cancels - abort	Trivial to identify comment-action. Difficult to identify 'abort' in signature since main action is at signature's end

`/** Many HTML components can have an <tt>accesskey<tt>`
` * attribute which defines a hot key for keyboard`
` * navigation. This method verifies that all the`
` * <tt>accesskey<tt> attributes on the specified page`
` * are unique. */`
`void assertAccessKeyAttrsUnique(HtmlPage page){ ... }`
	verifies - assert	Difficult to identify 'verifies' from comment since deep within comment and preceded by other actions such as 'have' and 'define'. Trivial signature-action identification

`/** Method called by the framework immediatelly after`
` * RRD update operation finishes. This method will`
` * synchronize in-memory cache with the disk content */`
`void afterUpdate(){ ... }`
	synchronize - update	Difficult to identify 'synchronize' from comment since preceded by incorrect actions such as 'called', 'update', and 'finishes'. Difficult to identify 'update' from signature because it does not begin signature

`/** You may decide to overload this or take the default`
` * and implement the functionality in your MapModel`
` * (implements MindMap). */`
`void load(File file){ ... }`
	- load	No main action found in comment. Trivial signature-action identification

`/** Respond to key presses. */`
`void keyPressed(KeyPressedEvent e){ ... }`
	respond -	Trivial comment-action identification. No signature action found

However, a naive approach that assumes these conventions will result in many incorrect mined word pairs.

Table I illustrates some of the key challenges to be addressed to obtain an accurate set of mined semantically-similar words through comment-code mappings. The second example shows how the main action from the signature (in this case, 'abort') is not always at the start of the method name, and similarly, the third and fourth examples show how the main action of the comment may be somewhere embedded in the comment block with many other earlier verbs detracting from its easy identification as the main action. In example 3, 'verifies' is the main action verb of the comment, but the earlier words 'have' and 'defines' are also verbs in the comment. In the fourth example, we need to filter out the first comment phrase as it is not a descriptive comment so we do not consider 'called' or 'update' or 'finishes' as the main action. The main action we identify automatically should be 'synchronize'.

Sometimes, there is no main action described by the comment or method name. These situations do not create opportunity for mining. In the fifth example, 'decide', 'overload', 'take', 'implement' and 'implements' are all verbs, but none of them describes the main action of the method. An automatic system needs to recognize and discard these cases so that incorrect semantically similar words (e.g., decide-load or overload-load or take-load) are not mined. In the sixth example, 'pressed' is a verb in the method name, but it is not describing the main action of the method, but instead describing a condition under which the method action is performed, 'when a key is pressed'. Without automatically identifying this, we would incorrectly mine that 'respond' and 'pressed' are semantically similar.

We summarize the main challenges in each of the three main phases of our approach.

Identifying Descriptive Leading Comments. Not all sentences or phrases in a leading comment block are descriptive, that is, describing the main intended action of the method. While they may contain a verb that appears to be a main action, they may be a 'Notes' comment that the developer left to remind themselves of something they need to do or be aware of, or a contextual comment that states that the method is 'called from X' or 'calls Y' or 'called when ...'. Furthermore, many of the non-descriptive comments indeed contain verbs, which could be mistaken as describing the main action of the method if these non-descriptive comments are not filtered out.

Extracting Documented Actions from Leading Comments. While finding the main action is straightforward when the leading comments begin with a verb in the present tense, many comments do not follow this convention. Sometimes, the main action verb is embedded in the comment. Sometimes, there are multiple verbs in the leading comment that could potentially be the main action. We first need to identify the verbs, and applying a part-of-speech tagger that was trained on English documents with full sentences starting with a subject noun phrase often results in mistagging the verbs when applied to comments due to the programmer convention of not using full sentences, or words used as verbs only in the programming sense. Typos and bad grammar also cause verbs to be mistaken as the main action.

Identifying Main Actions from Method Signatures. Even more difficult for a part-of-speech tagger trained on English documents is tagging words in a method signature, which has very dissimilar syntactic context. Many tagging mistakes ensue from the difference in syntactic context. In addition, some words in the programming domain differ from their typical uses in general. For example, 'fire' is typically a noun, but almost always a verb in computer science. While many methods start with a verb similar to leading comments because they are performing an action (e.g., sortList), the main action verb need not be the first word (e.g., dynamicColorUpdate). Thus, the location of the main action verb is not always the same. Sometimes, there is a verb, but it is not describing the main action of the method (e.g., eventDispatched, synchronizedList). Often, they are being used as an adjective. Also, sometimes the method name is just a noun phrase, while WordNet says one of the words can be a verb (e.g., size).

IV. Phases of Automation

A. Identifying Descriptive Leading Comments

We refer to the whole comment that precedes a documented method signature as a *leading comment block* and each component of the block that ends with a terminating special character such as '.', ':', ';' a *phrase* since often developers do not write full sentences in leading comments. Developers write several different kinds of leading comments. Sometimes the whole comment is of a certain kind, while other times, individual phrases are different kinds. Leading comment phrases can typically be categorized as notes, explanatory, contextual, evolutionary, conditional, or descriptive [17]. Notes are used to communicate pending tasks to either the developer or the team (e.g., TO DO: Fix ...), and sometimes used to refer to other parts of the code (e.g., See also,...). Explanatory comments explain the rationale behind some part of the code (e.g., We sort the list to gain better performance.). Contextual comments provide information about the context of the method (e.g., This method is called from ... or called by ... or calls...). Evolutionary comments describe information about the history of the method (e.g., This method was added in version 3.5.). Conditional comments specify conditions under which this method should be called (e.g., The caller must ... The first parameter should ...). Note that most of these other kinds of comments indeed contain verbs, but which do not describe the main intent of the method.

Since our goal is mining semantically-similar word pairs, we are gathering examples from a large corpus of potential comment-code mappings. Our goal is high precision in identifying descriptive leading comments such that we are not mapping action words from comment phrases that do not describe the intent of the method. Thus, as we identify descriptive leading comments as potential locations for comment-code mappings, recall is not a particular concern. That is, if we filter out some descriptive leading comments during our automatic identification of descriptive comments, this only decreases the number of potential mined comment-code mappings, but does not affect our results, given a large corpus. For other potential software engineering client tools that require higher recall of descriptive comments, our algorithm may need to be modified for increased recall.

Our automatic descriptive comment phrase identifier takes as input a leading comment block and outputs the sequence of phrases that we identify as descriptive phrases, in their original order. We execute the automatic descriptive comment phrase identifier on our corpus of documented methods. The heuristics

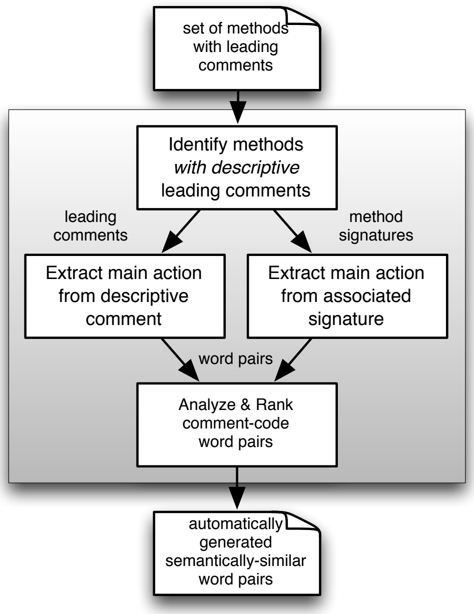
set of methods with leading comments
Identify methods *with descriptive* leading comments
leading comments
method signatures
Extract main action from descriptive comment
Extract main action from associated signature
word pairs
Analyze & Rank comment-code word pairs
automatically generated semantically-similar word pairs

Fig. 1. Process of automatically mining semantically-similar word pairs

are based on observations of programmer conventions in writing the different kinds of comments.

The first step is to eliminate leading comment blocks that are indicative of non-descriptive comments. We filter out the whole leading block when it satisfies either of the following properties:

- The comment begins with or is in complete all caps. Based on observations of programmer conventions, all caps is typically used to grab the attention of the reader, and typically signifies a notes comment (e.g., TODO, FIXME, HACK, REVISIT) [18], or an explanatory comment (e.g., NOTE:).
- The comment starts with a personal pronoun (e..g., We resort to avoid ..., You should ...). Developers commonly start notes and explanatory comments this way; this is our heuristic for filtering out these kinds of comments.

If the leading comment block is not eliminated, the second step is to examine each phrase in order to filter out non-descriptive comment phrases and then identify the main action from the first identified descriptive phrase. We filter out phrases as non-descriptive when they satisfy any of the following conditions:

- The phrase starts with an IF clause. We ignore any IF clause because it contains a condition under which to perform some action, not the action itself.
- The phrase contains any of the verb phrases: "returns", "called from", "performed by", "performed when", "uses". These phrases all indicate contextual information rather than descriptive information about the method, and signify a contextual phrase. Similarly, we ignore phrases with the verb 'see' as it mostly suggests a notes comment.

- The phrase contains a past tense verb. An example is "This method was added ..." From observational analysis, we discovered that past tense verbs usually indicate evolutionary comments, and are rarely describing the intent of the method.
- The phrase contains certain programming language keywords that typically signify an explanatory comment. Some examples such as the common term "overrides", which is typically followed by a message about a method's parenting methods, or "implements" do not explain the main intent of the method.

Otherwise, the only requirement we have to consider a comment phrase as descriptive is that it contain an action term either directly or indirectly followed by a noun term. The next section describes how we identify action and noun terms. Intuitively, our premise is that any useful English summary that describes the main intent of a method will possess a word order in which the verb precedes the object, indicating that an action is being performed on some object. We process each phrase in order until we find a suitable action description. Based on analyzing identifiers for several years, we observed that programmers typically write the main action within the first descriptive phrase of the leading comment.

B. Extracting Documented Actions from Leading Comments

Assuming that we are examining a phrase that has survived the filtering of non-descriptive phrases, we begin by identifying all potential verb-noun sequences in the phrase. We first identify the set of potential verbs (i.e., actions) and then determine those verbs that are followed by a word that could be a noun. These verb-noun sequences become our set of potential main actions documented by the descriptive comment.

We start by running Stanford's Log-Linear Part-of-Speech Tagger [19] on the current comment phrase under analysis. Each word in the method signature is tagged with a part of speech. Since the tagger is trained on English documents, sometimes the part of speech output is incorrect when applied to a method signature that does not follow the same sentence structure on which the tagger relies for tagging. Thus, we follow the tagger with a post processing step, particularly with the goal of identifying any words in the phrase that could be verbs in the computer science context, but not tagged as a verb by the Stanford tagger.

We assume that any word tagged as a verb by the Stanford tagger is a verb. However, there could be more words in the signature that act as verbs in the computer science context. We check if any of the non-verb tagged words in the phrase could be a verb in any sense by checking if it could be a possible verb according to WordNet. If so, we tag it as a potential verb. Sometimes, words used in software are not found in WordNet but when the prefix or suffix is ignored, they are found in WordNet, particularly as a verb. For example, 'unregister' and 'deregister', are not found in WordNet, but 'register' is a potential verb as indicated by WordNet. Similarly, 'setup' and 'backup' do not occur in WordNet, but 'set' and 'back' do occur as potential verbs after the suffix 'up' is removed. Thus, we process any word with one of a fixed set of prefixes or suffixes (e.g., re, dis, en, ex, out, re,...) and run the remaining partial word through WordNet to see if it can be a verb. If so, we tag it as a verb.

Sometimes putting the word into a full sentence can help the Stanford tagger to determine whether it is a verb. Thus, we put each remaining non-verb tagged word in the phrase into a template sentence, run the sentence through the Stanford tagger, and if the word is tagged as a verb in the sentence, we consider it a verb. We also process each of the partial words after removing the prefixes and suffixes in this way. We use three templates that we found useful for this purpose:

This method will WORD something.
This method will WORD.
This method WORD something.

Thus, we reevaluate words not tagged as verbs by the Stanford tagger using a staged postprocess which first utilizes WordNet to verify the questioned term as a known verb form and, for terms not found in the WordNet library, removes prefixes and suffixes, and places the terms into template sentences to determine potential as a verb. After we have all potential verbs in the phrase, we use the Stanford tagging output to determine which ones are followed immediately by a noun.

The next step is to determine which of these verb-noun sequences is most likely to describe the main action of the method. We process each verb-noun sequence in order, as programmer convention suggests that the main action is described earlier more often than later in a leading comment block. A naive approach would be to choose the first verb-noun phrase in the list as the main action; however, we have observed that this is not always the case for several reasons: mistagging of the verb, typos/bad grammar, and computer science specific conventions. Thus, we developed a set of rules that filter out verb-noun sequences that should not be considered to be the main action of the leading comment phrase, as they are encountered in order. The first verb-noun sequence that is not filtered out will be reported as the main action that the descriptive comment phrase is expressing.

- We always ignore verbs that are used too frequently and too vague to be useful for potential semantically similar word pairs. These includes verbs such as 'perform' and 'do', 'tries', 'test', 'determine', and linking verbs such as 'is', 'are', 'was', 'has. The linking verbs usually filter out evolutionary comment phrases (e.g., This method was developed for...).
- We always filter out verbs in the past participle form, because the verb usually is acting as an adjective and has been mistagged due to method signature context. For example, 'given' in "given database query string..." is an adjective, not a verb.
- We always filter out verbs in the past tense as they are also being used as adjectives in the method signature context. For example 'stacked' in "add stacked plot..." is an adjective.

- Under certain contexts, we ignore verbs in the base form. In particular, we keep as a candidate a verb-noun sequence where the verb is in base form and occurs within the first three words of the phrase or is preceded by a 'to'. For example, 'delete' in "delete the elected tracks" is a highly likely main action. Similarly, 'initialize' in "This method is called from within the constructor to initialize the form." However, again, there are situations where a word that is typically a verb is tagged as such, but it must be ignored because it acts as an adjective. For example, 'delete' in "attempt to set a delete frequency for the collector."
- Similarly, a verb in the third person singular form is kept as a candidate for the main action if it occurs in the first three words of the phrase, but filtered out otherwise. For example, 'populate' in "This method populate the..." is most likely a typo for 'populates'. An example that should be filtered out is 'update' in "...immediately before RRD update operation starts" where 'update' is acting as an adjective but mistagged as a verb.
- A gerund is kept as a candidate if it is preceded by a preposition but filtered out otherwise. For example, 'reading' in "convenience methods for reading a based64-encoded file" is most likely the main action. However, 'spanning' in "build a spanning tree" is being used as an adjective, and mistagged as a verb.

C. Identifying the Main Action from a Method Signature

Several approaches to tagging method names could have been used, including Abebe and Tonella's [20] template-based method name parser. However, we use a part-of-speech (POS) tagger [21] for method names that we've developed independently to address a larger set of programmer conventions.

First, through utilization of Samurai [22], we are able to efficiently break a method into its component words. Then, WordNet is implemented to quickly find possible word forms for the component pieces produced by Samurai. Finally, morphological rules are used for programming conventions because often software developers utilize their own terminology. For example, terms are often attached to the prefixes 'un' and 'de', such as in 'unregister' which is not a word found in WordNet. Our POS tagger then assigns all possible POS tags – from a set of 14 including singular nouns, base verbs, adjectives, past participles, etc. – to each word and chooses the most likely tag thereafter based on contextual clues. For example, the word 'fire' is normally used as a noun in general English, yet it is almost exclusively used as a verb in software development. By leveraging a large collection of compiled method and field names, we are able to determine the most likely tag. This process is useful for situations like 'dynamicColorsUpdate()', where we would find update as the verb based on similar clues. After the tagging process, we follow our design principle of being precise and cautious in implementing the following rules to extract the action term from the method signature.

- If a verb is found and determined to be in past tense or past participle, we do not use it as the main verb. If this verb is the first verb then our observations about method naming conventions suggest that it is not an action verb, rather it is typically adjectival and modifying the following noun (e.g., 'synchronizedList()'). Many of these names are just noun phrases and typically their action is only implicit. Additionally, past participle verbs at the end of a method name typically describe the situation under which the method is utilized, not what the method actually performs (e.g., 'mouseClicked()').
- We ignore method names beginning with linking verbs (e.g, 'is', 'has', 'can', etc) because they also do not describe the action being performed. Often these naming conventions are found for boolean methods in which the method name applies to the parameter(s) being passed (e.g., boolean canOpen(File f)).
- Methods starting with third-person singular verbs are also ignored because they are often related to checking if some condition holds, especially when they are of the boolean return type. For example, the method 'contains(...)' doesn't describe an action, but rather 'checks' if some parameter is contained.
- Gerunds are also filtered out in the case where they are followed by a past tense verb. For example, 'renderingCanceled()' hints at the state of the program, but doesn't describe what actions are performed.

D. Analyzing Comment-Code Word Pairs

A typical algorithmic NLP approach [24] to mining related words between two corpuses consists of taking an input word from one language and probabilistically determining the most mutually-dependent term from the parallel language based on all sentences of the input language that contain the input word.

In contrast, measuring relation between terms extracted from separate corpuses for the same purpose has less to do with probability than it does with frequency of occurrence. In our work, we are extracting terms from corpuses of leading comments and corresponding method signatures. A leading comment phrase and the method name are normally both expected to state the computational intent of the method, thus correctly identifying the main action from each provides a pair in which both terms are either the same word or different words expressing the same action. This claim holds true only if (1) the leading comment is both descriptive and accurate, (2) the method is well-named with a clear action term, and (3) our method accurately extracts the verbs from both cases. Rather than look at all translational pairs, we consider only pairs that we believe to be reflecting the main action of the method which considerably suppresses unnecessary pairs. Our hypothesis is that despite a small amount of error from the aforementioned three sources, incorrect pairs will not occur frequently enough to trump correct pairs, and thus a higher frequency indicates a stronger confidence that the pair is in fact semantically-similar.

V. Evaluation

Since there are multiple tasks comprising the automatic semantically-similar word finding technique, we designed our evaluation to answer the following five research questions via four studies:

1) How well does our automatic system determine descriptive comments?
2) How well does our automatic system identify the action word from a descriptive comment?
3) How accurate is the automatic extraction of the main action from a method signature?
4) How well do our word-pairs reflect reasonable semantically-similar words in computer science?
5) How well do we recall word-pairs from a human-annotated gold set?

Subjects, Variables, and Measures. The subjects in our study are methods from 36 open source Java programs across multiple domains and with different developers. In total, the projects are comprised of ∼3 million lines of code from 12,070 source files. In this dataset, there are 112,213 methods, with 20,199 methods documented by leading comments.

The independent variable is the multi-phase technique for mining semantically-similar words. The dependent variable is the effectiveness of each phase in terms of accuracy. We define accuracy for each study based on the objectives of that phase (see each study below).

Methodology. We ran each of the four phases (1: identifying methods with descriptive leading comments, 2: extracting action words in leading comments, 3: extracting action words in method signatures, and 4: analyzing and ranking mined semantically-similar word pairs) on the entire set of 36 Java projects. We engaged 6 human annotators, who had no knowledge of our techniques, were not authors on this paper, and who have extensive prior knowledge of the Java programming language. They performed several tasks to provide a gold set for each phase and human opinion of automatically mined, semantically-similar words.

We did not compare our results with Yang and Tan [16] because their technique analyzed each project on an individual basis, looking for word pairs that are semantically related within a common context. For example, they retrieved pairs including (save, do), which may be semantically related for the project jBidWatcher in which it was found, but which is difficult to justify as a universally semantically-related word pair to augment WordNet.

We now present the evaluation of each of the four phases of our automatic mining technique culminating with human opinion of our mined word pairs.

A. Study 1: Identifying Descriptive Leading Comments

Research Question: How well does our automatic system determine descriptive comments?

Procedure: From the set of 20,199 methods with leading comments in our dataset, we randomly selected 150 methods. We ensured that at least two-thirds of the comments were identified as descriptive by our technique in order to promote a reasonable sample size for the evaluation in the second study: main action extraction from descriptive comments.

TABLE II
Samples of results: Identifying descriptive comments

Automatic and Humans Agree

Leading Comment	Descriptive? (Y/N)
Called every n bytes, where n is defined by Settings	No
Switch to specified lex state.	Yes
Method used to create an about menu item for use in this application.	Yes
SWFActions interface	No

Automatic and Humans Disagree

Leading Comment	Descriptive? (Y/N) Automatic	Descriptive? (Y/N) Human
Torso twist to the left or right	No	Yes
If neither Date header nor Last-Modified header is present, current time is taken.	No	Yes
Client is current player; state changed from PLAY to PLAY1. (Dice has been rolled, or card played.) Update interface accordingly.	No	Yes
debug output for player trackers	Yes	No

Each of the six annotators was given 50 of these methods with leading comments. To limit the potential subjectivity, each of the 150 methods was examined by two annotators. In the case of a disagreement, an annotator who had not previously evaluated the method was utilized to break the impasse. This tie-break scenario was seen in 15 cases. We take the majority opinion as the annotation. For each annotated method, we asked:

1) Given the definition of a *descriptive leading comment*, do you believe the leading comment is descriptive, where we define a descriptive comment as ‘a comment that describes the intent of the code succinctly’?
2) If yes, identify the main action of the descriptive leading comment.

Results: Over all 150 methods, the annotators agreed with our automated technique in 131, or 87.33% of the cases. Of the 19 cases where the human annotators and automated technique disagreed, 14 were cases where the human annotators said the leading comment was descriptive, while our technique labeled it non-descriptive. Only 5 of the cases of disagreement between our automatic technique and the annotators were cases where we would have a high chance of incorrectly mining a descriptive action from a comment that in fact is believed to be non-descriptive of the main intent of the method.

Table II shows some examples of agreements and disagreements on descriptive comment classifications between the automated system and annotators. In the agreements, row 3, we see that the use of the preceding ‘to’ rule helped in identifying the action verb ‘create’. Contrastingly, our automated approach may have made a misjudgment for a case such as the 4th row of disagreements, where it seems ambiguous on whether debug was meant to be an action term or noun phrase. *These results suggest that our approach to identifying descriptive comments has fairly high accuracy, and the inaccuracy is mostly due*

TABLE III
SAMPLES OF RESULTS: EXTRACTING MAIN ACTION FROM COMMENT

Automatic and Humans Agree

Leading Comment	Extracted Action
This method is called from within the constructor to initialize the form. WARNING: Do NOT modify this code. The content of this method is always regenerated by the Form Editor.	initialize
Loads a tree description file.	loads
Immediately render the given VisualItem to the screen. This method bypasses the Display's offscreen buffer.	render
An abstract method that splits a line up into tokens. It should parse the line, and call addToken() to add syntax tokens to the token list. Then, it should return the initial token type for the next line.	splits

Automatic and Humans Disagree

Leading Comment	Extracted Action	
	Automatic	Human
Verifies that document.write() sends content to the correct destination (always somewhere in the body), and that script elements written to the document are executed in the correct order.	verifies	execute
Create a duplicate of this object.	create	duplicate
Regression test for bug 3017719: a 302 redirect should change the page url.	redirect	test
Drive the consumer by reading one tag	drive	reading

to mistakenly filtering out descriptive comments, which only leads to missed examples for mining.

B. Study 2: Extracting Actions from Leading Comments

Research Question: How well does our automatic system identify the action word from a descriptive comment?

Procedure: Of the 131 agreements on descriptive comment classifications between humans and automatic system, 118 of those were agreements on the comment being labeled as descriptive, while the remaining cases were agreements that the comments were non-descriptive. This study focused on evaluating the automated system's extraction of the main action from the 118 cases where there was agreement on the comment being descriptive.

When the annotators indicated that a comment was descriptive, they were then asked to indicate the word from the comment that they believed to be describing the main intent of the method. Thus, through our procedure from study 1, we ensured that we obtained at least 2 annotations of main actions from each of these comments.

Results: From the 118 cases where the annotators and automatic system agreed that the leading comment was in fact descriptive, there were 106 (or 89.93%) cases in which the automatic system agreed with at least one of the annotators. At least two annotators agreed on the same word for the main action in 93 comments. Of those 93 cases, they chose the same word as the automatic system for the main action 88 times (or 94.62%).

Table III shows some examples of agreements and disagreements on the main action word in descriptive comments between the automated system and annotators. In the agreements, it is clear that our automated approach correctly identified the action terms despite various tenses and phrasal patterns. In the disagreements, our automated approach struggled with tough cases. For example, in the 3rd row of the disagreements, our approach chose the action 'create' while the human annotators chose a more descriptive action of 'duplicate'. *Despite these tough cases, our automatic extraction of the main action from descriptive leading comments has high accuracy.*

TABLE IV
SAMPLES OF RESULTS: EXTRACTING MAIN ACTION FROM SIGNATURE

Automatic and Humans Agree

Method Signature	Extracted Action
void scheduleRedrawTimer()	schedule
int binarySearch(Object[] a, Object key, Comparator cp, int begin, int end)	search
void jsxFunction_collapse(boolean toStart)	collapse
void resetClick()	reset

Automatic and Humans Disagree

Method Signature	Extracted Action	
	Automatic	Human
void renderingCanceled()		canceled
SWFShape tagDefineShape2(int id, Rect outline)	define	tag
void makeTrade(int offering, int accepting)	make	trade
void fireToolStarted(DrawingView view)	fire	

C. Study 3: Identifying the Main Action from a Signature

Research Question: How accurate is the automatic extraction of the main action from a method signature?

Procedure: In total, 150 randomly selected method signatures were evaluated, each by two annotators. In the case of a disagreement, we broke the tie using critical judgement. The same 6 annotators also evaluated the signature action extraction process. Each annotator was given 50 of the 150 method signatures with any leading comments stripped off. They were asked to select the word from each signature that was equivalent to the main action, or none, if there was no term that they believed to be suitable.

Results: Over all 150 methods, there were 122 cases where both humans agreed on the action term from the signature, and 28 cases where they disagreed. For the 28 disagreements, the authors collaborated to distinguish the automated system's performance. Overall, the automated extraction identified the same main action of the signature as at least one of the annotators in 137 of 150 cases, for an accuracy of 91.33%.

Table IV presents a sampling of our results, with the original method signature and the automatically and human extracted main action words, for both agreement and disagreement cases. In row 3 of the disagreements, our automatic approach correctly identified a satisfactory action 'make'; however, the human annotators correctly identified a *more* descriptive term 'trade' which is more thoroughly indicative of the action. *This result indicates that the automatic extraction of main actions from method signatures has a high accuracy.*

D. Study 4: Generating Comment-Code Word Pairs as Semantically-Similar

Research Question: How well do our word-pairs reflect reasonable semantically-similar words in computer science?

TABLE V
ACCURACY OF PAIRS AT VARIOUS THRESHOLDS

Frequency	4+ Rated Pairs	Total Pairs	Accuracy
10+	26	33	78.79%
7–9	13	17	76.47%
5–6	23	47	48.94%
<5	Were not annotated		

Procedure: We gave 2 annotators – one from the aforementioned group and one author – a set of all 97 generated word pairs that had a frequency of occurrence of 5 or more via our automated mining system. The annotators were charged with the task of rating the pairs on a Likert scale from 1 to 5 based on their semantic-similarity, 1 representing no relation at all, 3 representing an unclear determination, and 5 representing inarguable semantic similarity. We used frequency thresholds to measure our approach's ability to find legitimate semantically-similar pairs. To study the usefulness of our approach in expanding lexical databases, we only considered word pairs that were not found to have semantic similarity in WordNet.

Results: Table V presents the results for a set of different frequency thresholds for a word pair to be considered semantically similar. For each threshold, we considered the number of annotated ratings that indicated positive semantic-similarity (i.e., a 4 or 5) against the total number of ratings for the pairs in that frequency range to determine its accuracy. Pairs at high frequency that did not receive a positive rating include (show, select), (modify, control), and (edit, present). In some situations, the poorly rated pairs were due to overloaded methods with identical comments. A sample of these 15 highly frequent pairs are shown in the top portion of Table VI. *Overall, these results indicate that the accuracy in semantic-similarity in our generated pairs indeed increases with frequency and implies more reliable mining with a larger dataset.*

Research Question: How well do we recall word-pairs from a human-annotated gold set?

Procedure: In addition to measuring the accuracy of our semantically-similar pairs, we compared our pairs to a gold set established by taking all 24 verb-pairs from various techniques tabulated in Table 4 of Sridhara et. al's [12] analysis. All verb-pairs taken were used in the evaluation of their work.

Results: In total, our technique successfully recalled 19 (or 79.17%) of the 24 pairs in the gold set. The bottom portion of Table VI shows 15 of these 19, since 4 pairs with high frequency are already listed in the top portion. The 5 missed cases from the gold set were (save, output), (flush, write), (reset, remove), (remove, cleanup), and (sort, compare). *These results indicate that our automatic method recalls word-pairs from the gold set in large proportions.*

E. Threats to Validity

We gathered a large collection of Java programs that would be reflective of code contexts and usages on a global level; however, the fact that only Java was used indicates that some programming language specific terms (e.g., 'malloc' in C/C++) may never be found. Additionally, the extensiveness of mining pairs relies directly on the quantity and variety of the

TABLE VI
SAMPLE OF AUTOMATICALLY MINED SEMANTICALLY-SIMILAR WORDS

Mined Word-pairs not in WordNet		
add - register	export - dump	fire - notify
find - search	finalize - clean	parse - read
handle - respond	fill - draw	clone - create
init - initialize	undo - restore	quit - close
paint - draw	display - print	reset - clear
Mined Pairs also in Gold Set Related Verb-pairs in Sridhara [12]		
display - show	determine - check	write - save
add - append	make - create	start - begin
remove - delete	compare - equal	store - add
locate - find	add - put	remove - clear
write - print	open - start	output - print

data set. For example, pair frequency could be relatively large on our data set, yet relatively low had we simply analyzed more projects or if we used an alternate data set of equal size.

As with all subjective tasks, it is possible that the human annotators did not correctly identify descriptive comments and actions, method signature actions, and semantically-similar word pairs. In some cases, a method's purpose or its leading comment description could be interpreted differently by separate annotators. It may also be possible that the annotators misinterpreted what were software-related terms or ideas that they were unfamiliar with. To limit this threat, we chose the annotators based on their experiences with software development, specifically in the Java domain.

VI. RELATED WORK

Beyond the most closely related work of Sridhara et.al [12] and Yang and Tan [16], several software maintenance tools have been developed that use some notion of synonyms. Shepherd et al. [2] developed FindConcept, a tool that expands search queries with synonyms to locate concerns more accurately in code. FindConcept obtains synonyms from WordNet, a lexical database of word relations that was manually constructed for English text.

A second maintenance tool that uses synonyms is iComment [25]. iComment is a tool that automatically expands queries with similar topic words to resolve inconsistencies between comments and code and therefore helps to automatically locate bugs. Their lexical database of word relations was automatically mined from the comments of two large programs. Their approach of automatically mining topic relations from code tacitly assumes that existing tools for semantic similarity on English text are insufficient when applied to software. To detect bugs, aComment [26] leverages semantically related words to identify comments that have similar meanings in order to check these comments against source code. aComment requires its users to manually specify synonyms and paraphrases. This is challenging because users must have domain knowledge about the target software, thus they are likely to miss important synonyms and paraphrases.

This work is somewhat related to strategies that automatically map comments to code by calculating a similarity measure between the set of words in both the method and the comment [27], [28]. Fluri et al. use a set-based similarity metric to explore how comments and code evolve over time [27]. Lawrie et al. compute the similarity between comments and code to assess code quality [28]. Their approach uses cosine similarity to calculate overlapping word usage between comments and methods, but does not map comment words to method words to calculate similarity.

There is also research that avoids explicitly handling semantic similarity by using an information retrieval (IR) technique that embeds semantic similarity information [29]. Latent Semantic Indexing is an IR technique that uses the co-occurrences of words in documents to discover hidden semantic relations between words. Since the technique is based on co-occurrences of words, the resulting word relations are not guaranteed to be semantically similar. LSI is also dependent on the set of documents (i.e., set of programs) from which the word co-occurrences are observed.

VII. Conclusions and Future Work

We demonstrate the many challenges in actually designing a system based on the simple observation that leading descriptive comments and the documented method names should indeed describe the same action. Our automatic miner of semantically-similar verbs has high accuracy in all its phases: 87% accuracy in identifying descriptive comments, with most inaccuracy due to filtering out descriptive comments which only leads to missed examples, 94% accuracy in extracting the correct main action word from a descriptive comment, 91% accuracy in identifying the main action word within the method signature, and finally agreeing with human opinion of the mined semantically-similar word pairs 78% of the time at a frequency threshold of 10 or more occurrences. Our hypothesis that accuracy in semantically-similar word pairs mined by our system increases with overall frequency is indeed justified.

We have some planned improvements that we have seen through our analysis of the results. Also, the work could be extended to make better choices when there are multiple actions with a conjunction, although this does not occur often. We also plan to investigate other potential opportunities for mining other word relations to improve client software tools.

Acknowledgements

We thank the whole Natural Language Program Analysis research group, particularly Giriprasad Sridhara, at University of Delaware.

References

[1] C. D. Manning, P. Raghavan, and H. Schuetze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[2] 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 '07: Proceedings of the 6th International Conference on Aspect-oriented Software Development*, 2007.

[3] C. Fellbaum, *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[4] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with Dora to expedite software maintenance," in *22nd IEEE International Conference on Automated Software Engineering (ASE)*, 2007.

[5] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[6] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings of the European Software Engineering Conference*, 2007.

[7] J. Krinke, "Identifying similar code with program dependence graphs," in *Eighth Working Conference on Reverse Engineering (WCRE)*, 2001.

[8] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *OOPSLA: Companion to the 21st ACM SIGPLAN*.

[9] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *27th International Conference on Software Engineering*, 2005.

[10] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE '06: Proceedings of the 28th Int. Conference on Software Engineering*, 2006.

[11] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *19th IEEE International Conference on Automated Software Engineering*, 2004.

[12] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker, "Identifying word relations in software: A comparative study of semantic similarity tools," in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 123–132.

[13] A. Budanitsky and G. Hirst, "Evaluating WordNet-based Measures of Lexical Semantic Relatedness," *Computational Linguistics*, vol. 32, no. 1, 2006.

[14] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Second Edition*. Prentice Hall, 2008.

[15] T. Pedersen, S. Banerjee, and S. Patwardhan, "Maximizing semantic relatedness to perform word sense disambiguation," University of Minnesota, Duluth, Tech. Rep., 2005.

[16] J. Yang and L. Tan, "Inferring semantically related words from software context," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, June 2012.

[17] G. Sridhara, "Automatic generation of descriptive summary comments for methods in object-oriented programs," Ph.D. dissertation, University of Delaware, Jan 2012.

[18] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "Todo or to bug: exploring how task annotations play a role in the work practices of software developers," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008.

[19] K. Toutanova, D. Klein, C. Manning, and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of HLT-NAACL 2003*, 2003, pp. 252–259.

[20] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10, 2010, pp. 156–159.

[21] S. Gupta, K. Vijay-Shanker, and L. Pollock, "Part-of-speech tagging of method names," U of Delaware, Tech. Rep. UD-CIS; 2013-002, February 2013.

[22] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *6th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2009.

[23] U. Germann. (2001) Aligned hansards of the 36th canadian parliament. [Online]. Available: http://www.isi.edu/natural-language/download/hansard/index.html

[24] C. D. Manning and H. Schuetze, *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

[25] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: Bugs or bad comments?*/," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. ACM, 2007, pp. 145–158.

[26] L. Tan, Y. Zhou, and Y. Padioleau, "aComment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *33rd International Conference on Software Engineering*.

[27] B. Fluri, M. Würsch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE)*, 2007.

[28] D. J. Lawrie, H. Feild, and D. Binkley, "Leveraged quality assessment using information retrieval techniques," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006.

[29] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *11th Working Conference on Reverse Engineering (WCRE'04)*, 2004.