Semantic Relation Based Expansion of Abbreviations

Yanjie Jiang
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
jiangyanjie@bit.edu.cn

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
Liuhui08@bit.edu.cn

Lu Zhang
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Beijing, China
zhanglu@sei.pku.edu.cn

ABSTRACT

Identifiers account for 70% of source code in terms of characters, and thus the quality of such identifiers is critical for program comprehension and software maintenance. For various reasons, however, many identifiers contain abbreviations, which reduces the readability and maintainability of source code. To this end, a number of approaches have been proposed to expand abbreviations in identifiers. However, such approaches are either inaccurate or confined to specific identifiers. To this end, in this paper we propose a generic and accurate approach to expand identifier abbreviations. The key insight of the approach is that abbreviations in the name of software entity e have great chance to find their full terms in names of software entities that are semantically related to e. Consequently, the proposed approach builds a knowledge graph to represent such entities and their relationships with e, and searches the graph for full terms. The optimal searching strategy for the graph could be learned automatically from a corpus of manually expanded abbreviations. We evaluate the proposed approach on nine well known open-source projects. Results of our k-fold evaluation suggest that the proposed approach improves the state of the art. It improves precision significantly from 29% to 85%, and recall from 29% to 77%. Evaluation results also suggest that the proposed generic approach is even better than the state-of-the-art parameter-specific approach in expanding parameter abbreviations, improving F_1 score significantly from 75% to 87%.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Abbreviation, Expansion, Knowledge Graph, Software Quality

ACM Reference Format:
Yanjie Jiang, Hui Liu, and Lu Zhang. 2019. Semantic Relation Based Expansion of Abbreviations. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338929

*corresponding author

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338929

1 INTRODUCTION

Identifiers are symbols used to identify uniquely a program entity in the source code [12]. According to the empirical study conducted by Deissenboeck et al.[15], identifiers account for the majority (70%) of source code in terms of characters. Consequently, the quality of such identifiers is crucial for the readability and maintainability of software applications[5][9].

Identifier abbreviations are popular [28]. Developers often use a short abbreviation to replace a long term or a sequence of terms. For example, they often use "e" to represent "exception" . However, such abbreviations have severe negative impact on program comprehension and IR-based software maintenance activities, e.g., concept location [31], software clustering [35], and recovery of traceability links [17][6].

To minimize the negative impact of abbreviations, a number of approaches have been proposed to expand abbreviations in source code. An intuitive and straightforward approach is to look up dictionaries, e.g., generic English dictionaries. For example, such approach may expand "ctx" into "context" because "context" is a dictionary word and dropping some characters from "context" results in "ctx". Such dictionary-based approaches are simple and straightforward, and thus are widely employed. However, there often exist multiple matching dictionary words for a single abbreviation (especially short abbreviation), and it is challenging to select the correct one from them. To improve the accuracy, researchers exploit the contexts of abbreviations, e.g., words within the same document or the same project. Exploiting such contexts has greatly improved the accuracy. However, the exploited contexts are often coarse-grained and ignore semantic relations, which has severe negative impact on the performance of abbreviation expansion[23].

To further improve the performance, in this paper we propose an automatic and generic approach to expanding abbreviations in identifiers. The key insight of the approach is that for an abbreviation in the name of software entity e, it is likely to find its full expansion in names of software entities that are semantically related to e. The rationale is that semantically related entities are likely to share some common concepts and thus their names may contain some common terms. Another characteristic of the proposed approach is that it is learning based. From a corpus of abbreviations and their full terms, the proposed approach learns automatically to prioritize matching heuristics and semantic relations. To the best of our knowledge, it is the first learning based approach to expanding identifier abbreviations.

We evaluate the proposed approach on nine well known open-source projects. Results of our k-fold evaluation suggest that the proposed approach significantly outperforms the state-of-the-art generic approaches. It improves precision from 29% to 85%, and recall from 29% to 77%. Evaluation on parameter abbreviations also suggests that the proposed generic approach outperforms the state-of-the-art parameter-specific approach in expanding parameter abbreviations, improving F_1 score significantly from 75% to 87%.

The paper makes the following contributions:

- A generic and accurate approach to expanding abbreviations in identifiers. The proposed approach differs from existing ones in that it exploits the semantic relations between software entities to expand identifier abbreviations. To the best of our knowledge, it also the first learning based approach to expanding identifier abbreviations.
- A prototype implementation and initial evaluation of the proposed approach. Evaluation results suggest that the proposed approach significantly improves the state of the art.

The rest of the paper is structured as follows. Section 2 introduces related research. Section 3 proposes the approach to expand abbreviation in identifiers. Section 4 presents an evaluation of the proposed approach on well-known open-source projects. Section 5 provides conclusions and future work.

2 RELATED WORK

2.1 Expansion of Identifier Abbreviations

Expansion of identifier abbreviations is to turn abbreviations in identifiers into dictionary words. Various approaches have been proposed to expanding identifier abbreviations.

Abbreviation dictionaries are frequently employed in abbreviation expansion [2]. An abbreviation dictionary contains a list of well-known abbreviations and their corresponding full terms. Abbreviation expansion based on abbreviation dictionaries often high-ly accurate. However, such abbreviation dictionaries are often constructed manually, which significantly limits the size of such dictionaries [11]. As a result, approaches that are completely based on such dictionaries can expand only a small number of abbreviations, resulting in low recall.

Generic English dictionaries are also employed for abbreviation expansion [37]. For a given abbreviation, generic-dictionary based approaches (*GD based approaches* for short) compare it against each term in the dictionaries, and return those that match the abbreviation according to predefined rules. The advantage of looking up expansion from generic English dictionaries is twofold. First, such dictionaries are ready for reuse, and developers do not have to construct such dictionaries again. Second, such dictionaries contain almost all possible English terms, and thus in most case we can find matching terms for given abbreviations. Because generic dictionaries are much larger than abbreviation dictionaries, *GD based approaches* can expand more abbreviations. However, it is quite often that for a given abbreviation there are a large number of matching terms from generic English dictionary, resulting in great challenge in selecting the correct one. To this end, more advanced approaches look for full terms from the context of abbreviations [30][26]. Such contexts include comments [27], and enclosing methods/documents/projects [25][14]. Corazza et al.[14], Lawrie et al.[27] and Madani et al.[30] suggest looking for full terms from comments of source code. However, developers rarely write comments. Lawrie et al.[25] suggests searching the enclosing methods whereas Hill et al.[22] propose a more complex approach to search enclosing methods, enclosing classes, and enclosing projects in order. Abdulrahman Alatawi et al.[3] leverage Bayesian unigram-based inference model to find full terms from source code. To the best of our knowledge, Abdulrahman Alatawi et al. [4] are the first to leverage natural language models in abbreviation expansion. Evaluation results suggest that the natural language models are highly accurate. These approaches have significantly improved the performance of abbreviation expansion. However, the exploited contexts are often coarse-grained and ignore semantic relations, which has severe negative impact on the performance of abbreviation expansion.

Besides the generic approaches introduced in the preceding paragraphs, Jiang et al.[23] propose a parameter-specific approach to expand parameter abbreviations only. By focusing on such a special subset of identifier abbreviations, the approach improves the performance (both precision and recall) significantly. However, it could not be applied to identifiers other than parameters.

Different matching algorithms are employed to search for potential expansions from given strings (sequences of words). Apostolio et al. [7] propose a matching algorithm where word *term* is regarded as a potential expansion of abbreviation *abbr* if *abbr* is a subsequence of *term*. If there are multiple potential expansions from given source strings, further strategies should be used to select one from potential expansions. Lawrie et al.[26] make suggestions only if the given abbreviation has a single potential expansion, and simply ignores the cases where multiple potential expansions are retrieved. In contrast, Hill et al.[22] and Carvalho et al.[13] sort such potential expansion according to their frequency, and recommend the top one. Lawrie et al.[25] choose the one who has the highest lexical similarity with the given abbreviation. Guerrouj et al.[21] and Corazza et al.[14] employ graph-based matching algorithms. Nodes of the graph represent letters and edges represent transformation costs. Based on the graph, they search for the shortest path from the initial letter of the abbreviation to anyone of its potential expansions with Dijkstra algorithm. The one with the shortest path is recommend as the most likely expansion. Jiang et al.[23] propose a series of heuristics to search for potential expansions. In case where multiple potential expansions exist, they select the shortest one.

The proposed approach differs from existing approaches in that it exploits the semantical relationship among software entities. It also differs from existing approaches in that it is the first learning based approach to abbreviation expansion.

2.2 Segmentation of Identifier Names

The segmentation of identifiers is to decompose identifiers into sequences of soft words[12]. It is closely related to the expansion of abbreviations because the result of segmentation may significantly influence the process of expansion.

If identifiers follow some common conventions such as Camel Case convention, the segmentation of such identifiers would be quite straightforward. For examples, we can segment identifiers

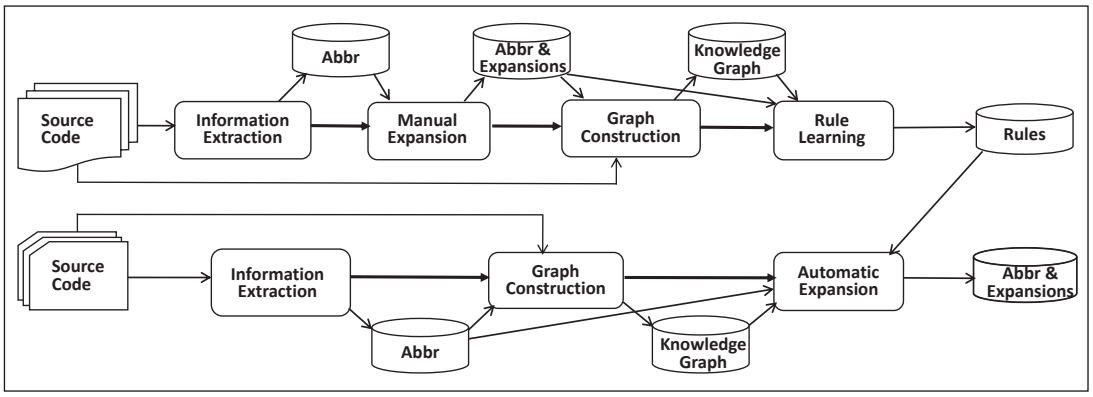

Figure 1: Overview of the Proposed Approach

according the positions of some characters, such as underscores, and capital characters[34]. However there still exist some identifiers that do not follow such naming conventions. To this end, researchers have proposed some complex and effective approaches to segment such identifiers.

Feild et al.[19] segment identifiers into constituent parts based on three word lists: a list of dictionary words, a list of well known abbreviations, and a list of stop words. First, they segment a given identifier into a sequence of hard words according to the positions of some characters Second, for each of the resulting hard words, if it appears in one of the word lists, it is regarded as a single soft word. Otherwise, they search for the longest prefix (or suffix) of the hard word that appears in one of the three lists. Enslen et al.[18] segment identifiers into sequences of terms based on two frequency tables: program-specific frequency table and global frequency table. Based on the two tables, this approach ranks possible splits of an identifier based on a scoring function. Lawrie et al.[26] proposed an approach named *GenTest*, and Carvalho et al.[13] proposed an approach named *IDSplitter*. Compared with the approach proposed by Enslen et al., they employ different scoring functions: *GenTest* computes scores with a series of metrics (e.g.,number_of_words, and co_occurrence), and *IDSplitter* simply counts the length of each term.

Guerrouj et al.[20] proposed an approach named *TIDIER*, and an enhanced version named *TRIS*[21]. *TIDIER* finds the best splitting based on a greedy search algorithm and the string edit distance. *TRIS* handles the segmentation of identifiers as an optimization problem and compares different segmentations and selects the one with the smallest cost as the result of segmentation.

Graph-based algorithms are also employed for the segmentation of identifiers. Corazza et al.[14] employ an approximate string matching techniques (called Baeaz-Yates and Perleberg)[8] to segment identifiers. Ashish Sureka[38] employs a recursive algorithm to segment camel-case identifiers. It splits an identifier into two substrings, noted as S_{left} and S_{right}. The optimal segmentation position is found based on a complex scoring function.

2.3 Knowledge Graph

To facilitate information revival, Google proposed Knowledge Graph in May 2012 [36]. Knowledge graph is a graph presenting entities and their semantical relations. Vertexes of knowledge graphs represent entities whereas edges present relations between the entities [33]. We may fuse information from a variety of data sources to construct comprehensive semantic network [32]. By constructing knowledge graphs, we can analyze complex relations between entities and thus improve the quality of services in different fields, e.g., recommender systems [24] and search engines [40].

In the field of software engineering, knowledge graphs have been successfully employed to facilitate bug resolution [39], to refine traceability links[16], and to improve API caveats accessibility [29]. Wang et al. [39] construct a bug knowledge graph based on their self-constructed bug knowledge. In the bug knowledge graph, entities are descriptions of bugs, commits, the class names and names of relevant developers; there are different edges between these entities, which represent different relations. Developers can search the bug knowledge graph to find accurate and comprehensive information of a software bug issue. Du et al. [16] construct a vulnerability knowledge graph. They firstly build CVE ontology, Maven ontology, and Github ontology, and then match these ontologies to form this vulnerability knowledge graph. It facilitates the vulnerability spreading analysis. Li et al. [29] construct an API caveats knowledge graph. They extract API caveat sentences from API documentations, and link these sentences to API entities to form this API caveats knowledge graph. It improves the API caveats accessibility, and developers can avoid many unexpected programming errors. All such successful applications of knowledge graphs suggest that knowledge graph has great potential in resolving software engineering tasks. This is one of the reasons why the proposed approach employs knowledge graphs.

3 APPROACH

In this section, we present an automatic approach to expanding abbreviations in identifiers. An overview of the proposed approach is presented in Section 3.1, and the details are presented in the following sections.

3.1 Overview

Fig. 1 presents the overview of the proposed approach. The proposed approach is divided into two phases: training phase (the upper part of Fig. 1) and testing phase (the lower part of Fig. 1). In the training phase, the proposed approach learns from a corpus of training data (abbreviations and their manually labeled full terms), which results in a sequence of rules for abbreviation expansion. The training phase works as follows:

(1) From a corpus of source code, it extracts identifiers, and developers manually expand abbreviations within the extracted identifiers.
(2) For each of the identifiers containing abbreviations, it builds automatically a knowledge graph to represent identifiers that are semantically related to it.
(3) Based on the knowledge graphs and the manual expansion, it employs an algorithm to learn a set of rules for abbreviation expansion.

With the resulting rules learned in the training phase, the proposed approach expands abbreviations automatically in the testing (predicting) phase as follows:

(1) First, it retrieves identifiers that contain abbreviations from the given application (source code).
(2) Second, it constructs a knowledge graph for each of the resulting identifiers.
(3) Finally, it leverages the expansion rules learned in the training phase to expand the abbreviations automatically.

Details of the proposed approach are presented in the following sections.

3.2 Software Entities and Semantic Relations

Identifiers are symbols used to identify uniquely a program entity in the source code. Such entities include classes, methods, variables, parameters, and fields. We represent all entities within a software application as a set E.

The proposed approach is based on the assumption that abbreviations within the name of an entity e_i have great chance to find their full terms from names of entities that are semantically related to e_i. Consequently, to expand abbreviations in the name of e_i, we should retrieve software entities (notated as $SE(e_i)$) that semantically related to e_i. Software entity e_j is added to $SE(e_i)$ if it is connected to e_i in one (or more) of the following ways:

- Inclusion. Software entity e_i is directly included by e_j, or the reverse. For example, if e_i is a class, all methods and fields within this class should be added to $SE(e_i)$. If e_i is a method, its enclosing class should be added to $SE(e_i)$.
- Assignment. e_i is assigned to e_j, or the reverse. The assignment includes not only the assignment operator ('='), but also assignments in broad sense, e.g., assignment of arguments to parameter.
- Inheritance. Software entity e_i is the superclass or subclass of e_j.
- Typing. Software entity e_j defines the type of e_i.
- Commenting. Although comments in source code dose not influence the semantics of programs, they are rich source of full terms.Consequently, we add comments explicitly associated with e_i to $SE(e_i)$ as well. For example, if e_i is a method, we include the comment that precedes the signature of e_i, but exclude comments within the method body.

3.3 Knowledge Graph of Software Entities

For software entity e, we automatically create a knowledge graph to represent its semantically related entities (i.e., $SE(e)$) as well as their semantic relationship with e.

A knowledge graph g for software entity e could be presented as a typed directed graph:

$$g(e) \quad = \quad < V(e), E(e), T(e) > \tag{1}$$

where $V(e)$ is a set of nodes, $E(e)$ presents directed edges between nodes, and $T(e)$ defines the types of the edges in $E(e)$. We also have

$$V(e) = SE(e) \cup e \tag{2}$$

$$E(e) = \{< e, v > | v \in SE(e)\} \tag{3}$$

because the knowledge graph presents the relationship between e and its semantically related entities (i.e., $SE(e)$).

Types of the edges are defined at finer granularity than those in Section 3.2. For example, the *inclusion* defined in Section 3.2 is further classified into *class-method*, *method-class*, *class-field*, *field-class*,*method-variable*, *variable-method*, *method-parameter*, *parameter-method*, and *variable-class*.

3.4 Segmentation of Identifiers

To extract abbreviations from identifiers, we split identifiers into sequences of soft words. Based on the assumption that identifiers follow the widely used Camel Case convention, we split such names as follows:

(1) First, we split an identifier into a sequence of hard terms according to underscores and numbers. For example, identifier "*v_trans2*" is split into < "*v*", "*trans*", "2" >.
(2) Second, we split an identifier (or a hard term resulted from the preceding segmentation) into a sequence of terms according to capital letters. Every capital letter is taken as a segmentation flag. An exception is the consecutive capital letters followed by lowercase ones (e.g.,URIValue). In this case, only the last capital letter is employed as a segmentation flag. Consequently, "URIValue" is split into "URI" and "Value".

Each of the resulting soft words is taken as an abbreviation if it does not appear in a generic English dictionary that is composed of full terms only.

3.5 Heuristics for Directed Expansion

For an given abbreviation (notated as *abbr*) within the name of an entity e_i, the proposed approach should try to find their full terms from names of entities that are semantically related to e_i (noted as $SE(e_i)$). While Section 3.7 discusses how the proposed approach decides the order in which entities in $SE(e_i)$ should be tried, this section presents a sequence of heuristics (noted as $SH =< H_1, H_2, H_3 >$) to find full terms of *abbr* from the name of a given entity $e_j \in SE(e_i)$. Such heuristics were proposed by Jiang et al. [23], and full explanations are available on their paper. However, for the sake of integrity of this paper, we give a brief introduction to such heuristics here.

H_1**: Acronym:** To replace a sequence of full terms in identifiers (and thus to shorten the identifiers), developers often coin abbreviations by concatenating the initial characters of such terms [10]. For example, they may coin "tf" to represent "TransformerFactory". Based on this observation, the first heuristic H_1 searches full terms from identifier id_j for abbreviation $abbr$ as follows: First, id_j is segmented into a sequence of terms, notated as T. Second, the initial characters of terms in T are concatenated, and the resulting string is represented as acr. If $abbr$ is identical to acr (case insensitive), terms in T are returned as the full terms for $abbr$.

H_2**: Prefix:** To shorten an identifier, developers often replace a lengthy term with a prefix of it. A well-known and widely used example is "str" that is frequently employed to represent "string". Based on this observation, the second heuristic H_2 searches full terms from identifier id_j for a given abbreviation $abbr$ as follows: First, it divides id_j into a sequence of terms. Second, for each term (noted as t) in the sequence, this heuristic compares t against $abbr$ to test whether $abbr$ is a prefix of it. If yes, term t is returned as a potential expansion of $abbr$. In case where the heuristic retrieves multiple potential expansions for the same abbreviation, it selects the shortest one.

H_3**: Dropped-Letters:** Dropping letters from lengthy terms is another way to coin abbreviations. For example, we may represent "index" with "idx" by dropping letters "n" and "e". Such abbreviations are often called dropped-letters [14][22]. Based on this observation, the third heuristic H_3 searches full terms from identifier id_j for a given abbreviation $abbr$ as follows: First, the heuristic divides id_j into a sequence of terms, noted as T. Second, it tests whether $abbr$ is a dropped-letters of any terms from T. If yes, such terms are marked as potential full terms. In case where the heuristic retrieves multiple potential expansions for the same abbreviation, it selects the shortest one.

3.6 Creation of Training Data

The proposed approach is learning based, and thus high-quality training data are indispensable. we create training data as follows. First, we download a corpus of source code from Github. Second, from the downloaded source code, we randomly extract a number of abbreviations, and represent such abbreviations as a list

$$abbrList = (< abbr_1, e_1 >, < abbr_2, e_2 >, ..., < abbr_n, e_n >) \quad (4)$$

where $abbr_i$ is an abbreviation contained in the identifier of software entity e_i. Third, we manually expand each of the abbreviations in $abbrList$. Notably, to reduce bias and inaccuracy, we need a group of developers to expand such abbreviations. They should discuss and vote if necessary to reach an agreement on each of the expansion. If they fail to reach an agreement on an abbreviation $abbr_i$, we should exclude this abbreviation from further analysis. The results of the manual expansion are represented as:

$$abbrExpans = \{< abbr_i, expan_i > | abbr_i \in abbrList\} \quad (5)$$

where $expan_i$ is the full expansion of $abbr_i$. Finally, we construct a knowledge graph g_i (as specified in Section 3.3) for each software entity e_i that is involved in $abbrList$. We also associate the knowledge g_i with the software entity e_i by a mapping $f_g : ETS \rightarrow KG$ where ETS is the involved software entities and KG are created knowledge graphs. As a result, we present the training data as:

$$TrData =< abbrExpans, f_g > \quad (6)$$

3.7 Rules for Expansion

As specified in Section 3.2 and Section 3.3, for abbreviations from the name of software entity e, the proposed approach should search for their full terms on the knowledge graph of e. We employ an expansion rule $r(e)$ to guide the abbreviation expansion. The rule (searching strategy) is essentially a sequence of search actions:

$$r(e) =< a_1, a_2, \ldots, a_n > \quad (7)$$

Where a_i is a search action. According the expansion rule $r(e)$, we should search for full terms according to the first search action a_1. If fail, we turn to the next action (and the followings if needed) until we succeed or no more actions are available. A search action is composed of a source and a matching heuristic:

$$a_i =< s_j, h_k > \quad (8)$$

$$s_j \in SR(e) \quad (9)$$

$$SR(e) = T(e) \cup Dict \quad (10)$$

$$h_k \in \{H_1, H_2, H_3\} \quad (11)$$

where $T(e)$ (as specified in Formula 1) defines the types of the edges in the knowledge graph of e, $Dict$ represents common abbreviation dictionaries, and $SR(e)$ represents all potential source for the search actions. s_j specifies what kind of edges (or abbreviation dictionary) should be exploited whereas h_k specifies which of the matching heuristics defined in Section 3.5 should be adopted. For example, a search action

$$< parameter - method, H_1 > \quad (12)$$

suggests that we should employ heuristic H_1 (Acronym) to find full terms from the name of the enclosing method.

The maximal number of search actions for element e equals $3 \times |SR(e)|$, where $|SR(e)|$ is the size of search space.

3.8 Learning Rules

As specified in Section 3.7, there is a large number ($3 \times |SR(e)|$) of search actions for element e. To specify the expansion rule $r(e)$ for this element, the key is to determine the order in which different search actions should be tried. To this end, in this section, we propose an algorithm to sort different search actions so that the most fruitful actions are tried first.

Based on training data $TrData$ (as defined in Formula 6), we learn to sort search actions for e as follows.

- As defined in Equation 4, $abbrList$ contains all abbreviations as well as their associated software entities. From the list, we retrieve software elements (as well as associated abbreviations) that are of the same type as e. For example, if e is a method, we retrieve methods only from the list. The resulting list is noted as:

$$abbrList(e) = (< abbr_{x_1}, e_{x_1} >, \ldots, < abbr_{x_m}, e_{x_m} >) \quad (13)$$

- We apply each of the possible search actions to expand the abbreviations in $abbrList(e)$.

- We sort all of the possible search actions according to their priority (in descending order) that is defined as follows:

$$priority(a_i, e, abbrList) = \frac{p(a_i, e, abbrList(e)}{1 + e^{-TP(a_i, e, abbrList(e))}} \quad (14)$$

where a_i is a search action, $TP(a_i, e, abbrList(e))$ is the number of abbreviations in training data $abbrList(e)$ that are successfully expanded by action a_i. $p(a_i, e, abbrList(e))$ is the precision of the expansion guided by a_i on training data $abbrList(e)$.
- The resulting sequence of the possible search actions is taken as the expansion rule $r(e)$.

3.9 Automatic Expansion

Algorithm 1 Abbreviation Expansion

Input: *abbr* % abbreviation to be expanded
e % associated software entity
$g(e)$ % knowledge graph of e
$r(e)$ % expansion rule for e
Output: *expan* % full expansion of *abbr*

```
1:  for each a in r(e) do
2:      names ← RetrieveIdentifiers(e, g(e), a.s)
3:      expans ← ∅
4:      for each name in names do
5:          potentialExpan ← getExpansions(abbr, name, a.h)
6:          expans ← expans ∪ potentialExpan
7:      end for
8:      if expans = NULL then
9:          continue
10:     else
11:         expans ← SortAndSelect(expans)
12:         return expan
13:     end if
14: end for
15: return NULL
```

As suggested by Fig. 1, the last step of the proposed approach is automatic expansion that expands abbreviations automatically based on learned expansion rules. The algorithm of automatic expansion is presented in Algorithm 1. The input of the algorithm is an abbreviation (*abbr*), its associated software entity (e), the knowledge graph $g(e)$, and the learned expansion rule $r(e)$. The output is the resulting full expansion of the abbreviation *abbr*.

The algorithm works as follows:

- First, it retrieves the first search action a from $r(e)$ (Line 1);
- Second, according to the edges specified by the $a.s$, it retrieve identifiers (*names*) of entities that are semantically related to e (Line 2);
- Third, for each identifier in *names*, it retrieves full expansions for *abbr* according to matching heuristic $a.h$, and appends the resulting expansions to dataset *expans* (Lines 3-7);
- Fourth, if the expansion guided by action a fails (i.e., *expans* = *NULL*), it tries the next action in $r(e)$ (Lines 8-9) and begins the next iteration. Otherwise, it goes to the next step;
- From all of the potential expansions in *expans*, it selects the one with the highest frequency (Line 11). That is if exp_1 appears 3 times in *expans* whereas exp_2 appears twice, exp_1 is preferred. If exp_1 and exp_2 have the same frequency, the shorter one is selected. If exp_1 and exp_2 share the same frequency and the the same length, it randomly selects one of them.
- Finally, if all of the actions in $r(e)$ fail, it returns *NULL* (Line 15), suggesting that it fails to expand the abbreviation *abbr*.

4 EVALUATION

In this section, we evaluate the proposed approach (noted as *KgExpander*) on nine well known open-source applications.

4.1 Research Questions

- **RQ1**: How often can abbreviations in the name of software entity e find their full terms in names of software entities that are semantically related to e ?
- **RQ2**: Is the proposed approach accurate in expanding abbreviation? Does it outperform the state-of-the-art approaches?
- **RQ3:** Does the proposed approach outperform parameter-specific approaches in expanding parameter abbreviation? If yes, to what extent and why?
- **RQ4**: How does the abbreviations' length influence the performance of abbreviation expansion?
- **RQ5:** Dose the prioritization of potential actions influence the performance of the proposed approach? If yes, to what extent?
- **RQ6**: Is the proposed approach scalable?

The proposed approach is based on the assumption that abbreviations in the name of software entity e have great chance to find their full terms in names of software entities that are semantically related to e. Answering research question **RQ1** would validate the assumption.

Research question **RQ2** investigates the performance (e.g., precision and recall) of the proposed approach in expanding abbreviations in comparison with the state-of-the-art approaches. To answer this question, we compare the proposed approach against *Linsen* [14] because *Linsen* is accurate, well-known, and represents the state-of-the-art. Besides that, the implementation of *Linsen* is publicly available, which facilitates the comparison. Answering this research question would reveal to what extent the proposed approach improves the state of the art.

Besides generic approaches (like *Linsen*) that is designed to expand abbreviations from different kinds of identifiers, a parameter-specific approach is proposed by Jiang et al. [23] (we call it *parExpander* for convenience). Although *parExpander* is confined to abbreviations in parameters (actual or formal parameters), it is highly accurate [23]. Research question **RQ3** investigate whether the proposed generic approach can outperform *parExpander* even if only parameter abbreviations are concerned.

RQ4 concerns the impact of abbreviations' length. As suggested by Jiang et al. [23], the length of abbreviations does influence the performance of existing approaches. Answering this question helps to reveal whether our approach is also influenced by abbreviations' length. **RQ5** concerns the impact of the prioritization of potential actions. As specified in Section 3.8, the prioritization is learned from a corpus of abbreviations (accompanied with their corresponding expansions) that are manually expanded. Answering this question

Table 1: Comparison among Different Searching Scopes

Application	Knowledge Graph		Enclosing Method		Enclosing File		Enclosing Project	
	$P_{presence}$	size	$P_{presence}$	size	$P_{presence}$	size	$P_{presence}$	size
DB-Manager	55%	5	57%	24	68%	62	72%	266
Batik	84%	11	72%	25	78%	100	88%	2146
Portecle	75%	11	61%	35	72%	160	79%	711
PDFsam	46%	8	37%	24	49%	58	62%	998
Retrofit	25%	7	26%	20	37%	89	40%	728
Bootique	54%	8	53%	16	56%	44	63%	680
CheckStyle	44%	13	43%	22	51%	97	69%	1831
Maven	77%	10	74%	24	76%	75	80%	1443
FileBot	62%	9	56%	21	62%	110	75%	1566
Average	58%	9	53%	24	60%	88	69%	1152

would reveal the value of the learning phase. **RQ6** concerns the scalability of the proposed approach, i.e., whether the proposed approach can work on large projects.

4.2 Subject Applications

We reuse the subject applications employed by Jiang et al. [23]. We reuse such subject applications because of the following reasons. First, all of the subject applications are open-source applications whose source code is publicly available, which facilitates other researchers to repeat the evaluation. Second, such applications have great diversity. They are from different domains, and developed by different programmers. The size of such projects varies dramatically from 1,992 LOC to 384,906 LOC, and such projects cover single-developer small projects as well as large projects where a large number of contributors cooperate extensively. Besides that, the dataset includes both libraries and applications.

4.3 Process

The process of the evaluation is composed of three phases: sampling, manual expansion, and k-fold cross validation on the resulting dataset. These phases are explained in detail in the following paragraphs.

To create a dataset for evaluation, we download nine applications from Github based on the popularity. Then, we randomly pick up 200 identifier abbreviations from each of the involved subject applications (as introduced in the preceding section 3.6). We follow the sampling strategy proposed by Jiang et al. [23] to reduce potential bias. The strategy excludes identical identifiers, i.e., identifiers in the resulting dataset are lexically different from each other. Notably, for the smallest project DB-Manag-er, the number (53) of abbreviations from unique identifiers is fewer than 200, and thus we include all such abbreviations.

Manual expansion of sampled abbreviations is conducted by three software developers. They are master level students, and are familiar with Java. However, they are not aware of the proposed approach. According to the information from source code, they recommend an expansion for each of the abbreviations. If they come up with different expansions for the same abbreviation, they are requested to discuss together and reach an agreement before the final expansion is confirmed. The resulting dataset is publicly available at https://github.com/4DataShare/AbbExpansion.

The k-fold (k=9) cross-validation on the resulting dataset is conducted as follows. On each fold, a single application (out of the nine subject applications) is used as the testing subject (noted as *testingApp*) whereas the others are used as training subjects (noted as *trainingApps*). Each of the subject applications is used as the testing subject for once. Each fold of the evaluation follows the following process:

- **Step1**, manually expanded abbreviations from *trainingApps* are collected as training data, noted as *trainingData*. Other sampled abbreviations are collected as *testingData*;
- **Step2**, the proposed approach learns rules with *trainingData*;
- **Step3**, we apply the proposed approach and *Linsen* to *testingData*, respectively.
- **Step4**, We compute the performance (i.e., precision and recall) of the evaluated approaches. An expansion of a given abbreviation is correct if and only if the expansion is identical to the manual expansion stored in the testing dataset.

As specified in Section 3.7, abbreviation dictionaries are useful in abbreviation expansion, and thus they are exploited by the proposed approach and existing approaches like *Linsen* and *ParExpander*. In the evaluation, we employ two publicly available dictionaries: a generic abbreviation dictionaries [14] and a computer-specific abbreviation dictionary [1]. For fair comparison, all evaluated approaches use the same dictionaries.

4.4 RQ1:Semantic Relations are crucial for Abbreviation Expansion

To answer research question **RQ1**, we manually expand 1,653 abbreviations in identifiers, and investigate where their full expansions could be found (The process of manual expansion is identical to Section 4.3). The key of the proposed approach is to construct a searching scope for a given abbreviations according to semantic relation. In contrast, existing approaches [25][14][13][22] often construct the searching scope according to distance, i.e., words within the enclosing methods, the enclosing documents or the enclosing projects. We employ the following two metrics to measure how good such search scopes are. The first is the size of the scopes, i,e., the average number of unique terms within the scopes. The larger the scope is, the more challenging to pick up the right ones. Consequently, the smaller the scope is, the better it is. The second is the possibility that the search scope contains the correct expansion

(noted as $P_{presence}$):

$$P_{presence} = \frac{No.\ of\ scopes\ containing\ expansions}{No.\ of\ scopes} \quad (15)$$

Greater $P_{presence}$ suggests greater chance to find full terms from the given scope. Evaluation results of different search strategies are presented in Table 1. The first column presents subject applications. Columns 2-3 present the metrics of the proposed searching scopes, i.e., knowledge graphs. Columns 4-9 presents the metrics of distance based searching scopes. Notably, abbreviation dictionaries employed by the proposed approach (and existing approaches as well) is not counted in by the searching scopes. These dictionaries often serve as supplements to different searching scopes, and thus they can collaborate with different searching strategies and different searching scopes [14].

From the table, we made the following observations:

- First, all of the searching scopes have great chance (varying from 53% to 69%) to contain the correct expansions of abbreviations. It may suggest that it is potentially fruitful to search full terms within such scopes. That is one of the reasons for the success of existing approaches, e.g., *Linsen*.
- Second, the average size (nine unique words) of knowledge graphs is significantly smaller than that of other searching scopes. It is only 37.5%(=9/24), 10.2%(=9/88), and 0.8%(=9/1152) of the sizes of enclosing methods, enclosing files, and enclosing projects, respectively. Considering that $P_{presence}$ of knowledge graphs is comparable to other searching scopes (even greater than that of enclosing methods), the significantly smaller size of knowledge graphs may suggest that searching full terms in knowledge graphs could be significantly more accurate (and thus more fruitful).

From the preceding analysis, we conclude that abbreviations in identifiers have great chance to find their full terms in semantic based searching scopes (i.e., knowledge graphes) as well as distance based searching scopes. However, knowledge graphes are significantly smaller than distance based searching scopes, and thus it will be much more accurate to search for full terms in knowledge graphs than in distance based searching scopes.

4.5 RQ2:Improving the State of the Art

To answer research question **RQ2**, we compare the proposed approach (*KgExpander*) against the state-of-the-art approach (*Linsen* and *BingramExpander*). Evaluation results are presented in Table 2. The first column of Table 2 presents the subject applications. The second to the fifth columns present the results of the proposed approach, the sixth to the ninth columns present the results of *Linsen*, and the tenth to the thirteenth columns present the results of *BingramExpander*. The second, the sixth, and the tenth columns present the number of abbreviations that are successfully expanded by the proposed approach, *Linsen*, and *BingramExpander*, respectively. The third, the seventh, and the eleventh columns pres-ent the incorrect expansions made by the proposed approach, *Linsen*, and *BingramExpander*, respectively. Notably, the precision and recall of *Linsen* are always the same because it expands all abbreviations it encounters. From the table, we made the following observations:

- First, the proposed approach is accurate. Its precision varies from 75% to 91%, with an average of 85%. High precision suggests that it makes few mistakes in abbreviation expansion, and thus it is reliable.
- Second, the proposed approach successfully expands most (77%) of abbreviations in identifiers. Such high recall indicates high serviceability of the proposed approach.
- Third, the proposed approach significantly outperforms the state-of-the-art approaches. Compared against *Linsen*, it improves the precision and recall by 193% = (85%-29%)/29% and 166% =(77%-29%)/29%, respectively. Compared against *BigramExpander*, it improves the precision and recall by 98% =(85%-43%)/43% and 148%=(77%-31%)/31%, respectively.
- Finally, the proposed approach outperforms *Linsen* and *Bigram−Expander* on every subject application. For the *Linsen*, the smallest improvement in precision and recall is 42% and 31%, respectively. For the *BigramExpander*, the smallest improvement in precision and recall is 27% and 34%, respectively.

We conclude from the preceding analysis that the proposed approach is accurate and it can expand most of the abbreviations in source code. We also conclude that it significantly outperforms the state-of-the-art approach *Linsen*.

4.6 RQ3:Expansion of Parameter Abbreviations

Although *parExpander* proposed by Jiang et al. [23] is confined to parameter, it is highly accurate. In this section, we investigate whether the proposed approach is comparable to (or even better than) *parExpander* in expanding parameter abbreviations.

Notably, only 25%(=416/1,653) of the abbreviations in our study are from parameters (including actual parameters and formal parameters). It suggests that the majority (75%) of the abbreviations are from non-parameters, and they could not be expanded by parameter-specific approaches like *parExpander*.

We compare the proposed approach against *parExpander* on the 416 parameter abbreviations that *parExpander* was intentionally designed to expand. Results are presented in Table 3. The first column of Table 3 presents the subject applications. The second to the fourth columns present the results of the proposed approach whereas the other columns present the results of *parExpander*. From this table, we make the following observations:

- First, *parExpander* is highly accurate in expanding parameter abbreviations, which is in consistent with previous study [23]. Its average precision is up to 96% whereas its average recall is up to 61%. We also observe that it performs well on each of the subject applications.
- Second, the proposed approach outperforms *parExpander* concerning F_1 score. It improves the average F_1 score from 75% to 87%, and the improvement is up to 16%=(87%-75%)/75%.
- Third, the proposed approach improves recall significantly. The average recall of the proposed approach is 85%, significantly greater than that (61%) of *parExpander*. The improvement is up to 39%=(85%-61%)/61%.
- Finally, the precision (90%) of the proposed is comparable to that (96%) of *parExpander* even if only parameter abbreviations are concerned.

Table 2: Comparison Against State-of-the-art Approach

Applications	KgExpander				Linsen				BigramExpander			
	Correct Expansion	Incorrect Expansion	Precision	Recall	Correct Expansion	Incorrect Expansion	Precision	Recall	Correct Expansion	Incorrect Expansion	Precision	Recall
DB-Manager	42	5	89%	79%	25	28	47%	47%	13	13	50%	25%
Batik	172	20	90%	86%	57	143	28%	28%	56	94	37%	28%
Portecle	163	17	91%	82%	55	145	28%	28%	78	94	45%	39%
PDFsam	139	47	75%	70%	41	159	21%	21%	41	81	34%	21%
Retrofit	128	34	79%	64%	58	142	29%	29%	24	82	23%	12%
Bootique	147	35	81%	74%	48	152	24%	24%	80	69	54%	40%
CheckStyle	142	32	82%	71%	79	121	40%	40%	60	85	41%	30%
Maven	175	21	89%	88%	57	143	28%	28%	93	72	56%	47%
FileBot	172	16	91%	86%	60	140	30%	30%	72	75	49%	36%
Total	1280	227	85%	77%	480	1173	29%	29%	517	665	43%	31%

Table 3: Expansion of Parameter Abbreviations

Application	KgExpander			parExpander		
	Precision	Recall	F_1	Precision	Recall	F_1
DB-Manager	100%	88%	94%	83%	63%	72%
Batik	98%	95%	96%	96%	50%	66%
Portecle	94%	84%	89%	94%	46%	62%
PDFsam	77%	74%	75%	89%	59%	71%
Retrofit	67%	67%	67 %	100%	27%	43%
Bootique	76%	73%	74%	95%	53%	68%
CheckStyle	92%	89%	90%	100%	93%	96%
Maven	95%	92%	93%	100%	72%	84%
FileBot	95%	89%	92%	97%	67%	79%
Average	90%	85%	**87%**	96%	61%	**75%**

We also investigate the reasons for the reduction in precision. We find that the proposed approach frequently expands parameter abbreviations that *parExpander* refuses to expand (noted as *refused abbreviations*). *parExpander* refuses to expand parameter abbreviations frequently (with a probability of 33%=138/416) because its searching scope is rather small and such scope often fails to contain the full expansions. In contrast, the proposed approach exploits more semantic relations, and thus has greater chance to find the full expansions. We also notice that the proposed approach is less accurate on such *refused abbreviations* than on other parameter abbreviations: the precision is reduced from 96% to 69%. As a result, the less accurate expansion of such prevalent *refused abbreviations* leads to moderate reduction in the average precision of the proposed approach. On the other side, however, the proposed approach successfully expands 85 out of the 138 refused abbreviations, and thus improves recall significantly from 61% to 85%.

We conclude from the preceding analysis that the proposed approach outperforms *parExpander* in expanding parameter abbreviations, leading to significantly improved recall.

4.7 RQ4: Impact of Abbreviations' Length

As suggested by existing studies [23], length of abbreviations has significant impact on the performance of *Linsen*, and the shorter abbreviations are usually more challenging to expand. To this end, we investigate whether the performance of the proposed approach

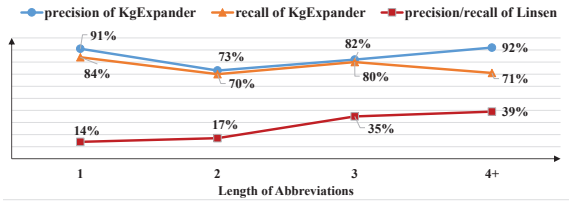

Figure 2: Impact of Abbreviations' Length

is significantly influenced by abbreviations' length. Evaluation results are present in Fig 2. Notable, *Linsen* tries to expand every abbreviation, and thus its recall is always equal to its precision. From Fig 2, we make the following observations:

- First, it is challenging for state-of-the-art approaches like *Linsen* to expand short abbreviations. Its precision (and recall as well) reduces dramatically from 35% to 17% when the length of abbreviations reduces from three characters to two characters. It even further reduces to 14% when the length reduces to one character.
- Second, the proposed approach works well for short abbreviations, especially single-character abbreviations. To reveal the reason for such high performance, we manually analyze such single-character abbreviations. Analysis results suggest that most (99.25%) of such abbreviations are acronyms (initial characters), e.g., "*s*" standing for "*string*". Such short acronyms are difficult to expand correctly because they may match a large number of potential full terms. For example, "s" could be abbreviation of any term beginning with 's'. The proposed approach succeeds because it limits the size of searching scopes, i.e., it compares the short abbreviations against a small number of terms from semantically related entities. As a result, the number of matching terms is significantly reduced, and the accuracy is guaranteed.
- Third, the proposed approach fails to expand a significant part (30%) of the two-character abbreviation. We manually analyze the failed cases to figure out the major reason for the failure. We find that 62.5%(=55/88) of the involved abbreviations should be expanded into two words but are expanded improperly by our approach into a single word. A typical example is "sw". The approach expands it into "software" whereas it stands for "string writer".

Table 4: Influence of Prioritization

Application	With Prioritization		Without Prioritization	
	Precision	Recall	Precision	Recall
DB-Manager	89%	79%	87%	77%
Batik	90%	86%	83%	80%
Portecle	91%	82%	86%	78%
PDFsam	75%	70%	74%	69%
Retrofit	79%	64%	78%	64%
Bootique	81%	74%	78%	71%
CheckStyle	82%	71%	78%	68%
Maven	89%	88%	86%	85%
FileBot	91%	86%	81%	76%
Total	**85%**	**77%**	**81%**	**74%**

We conclude from the preceding analysis that the length of abbreviations has less impact on our approach than *Linsen*, and our approach works well on short abbreviations, especially one-character abbreviations.

4.8 RQ5: Influence of Prioritization

In Section 3.7, the proposed approach learns to prioritize potential search actions based on a manually created training dataset. To investigate the impact of the prioritization, we repeat the evaluation but ignoring the learned prioritization, i.e., the approach looks for all potential expansions with all of the available search actions, and selects the most likely one as specified on Line 11 of Algorithm 1.

Evaluation results are presented in Table 4. From this table, we make the following observations:

- The prioritization improves the performance of the proposed approach in both precision and recall. It increases the precision from 81% to 85%, and recall from 74% to 77%.
- The prioritization takes effect on each of the subject applications. On each of the nine applications, the default setting (with prioritization) outperforms its counterpart (without prioritization).

During the evaluation, we also observe some fruitful actions for different software entities. For example, the most fruitful action for local variables is Type-H_2, and the most fruitful action for class names is superclass-H_2.

We conclude from the preceding analysis that the prioritization helps to improve the performance.

4.9 RQ6: Scalability

We investigate the scalability of the proposed approach by analyzing the relationship between execution time of the proposed approach and the size of testing projects. The evaluation is conducted on a personal computer with Intel Core i7-6700, 16GB RAM, and Windows 7. Notably, the execution time includes training time (as specific in Section 3.8). Based on the results, we make the following observations:

- First, the execution time increases when the size of projects increases;
- Second, the proposed approach is efficient, takeing around 5 minutes only to parse and expand 4,701 abbreviations in the largest project (CheckStyle) that contains more than 38 thousands of lines of source code.
- Third, disabling learning increases testing time significantly by 46% and decreases training time slightly by 3.5%.

We conclude from the preceding analysis that the proposed approach is scalable and could be applied to large projects.

4.10 Threats to Validity

A threat to construct validity is that the dataset used in the evaluation is constructed manually by three students instead of the original developers of the projects. They may make mistakes in the manual expansion because they lack system knowledge. Such mistakes may make the evaluation inaccurate. To reduce the threat, we ask the three participants to work together and exclude expansions where they cannot reach an agreement.

Another threat to validity concerning the manual golden-set creation is the bias to or against evaluated approaches. If the participants know in advance how the proposed approach works, they may expand the abbreviations as the proposed approach does, which may seriously bias the evaluation results. To reduce the threat, we exclude the authors of the paper from the manual creation, and recruit students who are not aware of the proposed approach.

A threat to the external validity is that only 9 projects and 1,653 abbreviations are involved in the evaluation. The limited number of involved projects and abbreviations may threaten the generality of the conclusions, i.e., the extent to which such conclusions can be generalized to other situations (other projects and abbreviations). Special characters of such projects may bias the conclusions, e.g., the native language of the developers, the style guide applied by the project, whether the project does code review, whether it is a single developer project or one with many developers collaborating, and whether it is a library or an application. To reduce the threat, we select such projects that cover diverse characters (as shown in Table 1). Evaluation results suggest that the proposed approach is accurate on each of the subject applications. To further improve the generality of the conclusions in future, however, evaluation on more projects and more abbreviations should be conducted.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we propose an automatic approach to expanding abbreviations in identifiers by exploiting the semantic relations among software entities. The key insight of the approach is that the full terms of an abbreviation (*abbr*) in the identifier of software entity e are likely to appear in the names of software entities that are semantically related to e. Consequently, we build a knowledge graph for e, presenting its semantically related entities (and their names) as well their relationship with e. Based on this knowledge graph, we search for full terms for *abbr* with a sequence of heuristics. To investigate what kind of semantical relations and heuristics are more reliable and potentially more fruitful, we prioritize potential search actions with a manually created training dataset. We evaluate the proposed approach on open-source projects. Results of the k-fold cross-validation suggest that the proposed approach is accurate, and it improves the state of the art significantly.

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China under Grant No.: 61690205, 61772071,61529201

REFERENCES

[1] [n. d.]. https://www.webpages.uidaho.edu/~jory/babel.html.

[2] Eytan Adar. 2004. SaRAD: A simple and robust abbreviation dictionary. *Bioinformatics* 20, 4 (2004), 527–533.

[3] Abdulrahman Alatawi, Weifeng Xu, and Dianxiang Xu. 2017. Bayesian Unigram-Based Inference for Expanding Abbreviations in Source Code. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 543–550.

[4] Abdulrahman Alatawi, Weifeng Xu, and Jie Yan. 2018. The Expansion of Source Code Abbreviations Using a Language Model. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. IEEE, 370–375.

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[6] Giuliano Antoniol, Gerardo Canfora, A De Lucia, and G Casazza. 2000. Information retrieval models for recovering traceability links between code and documentation. In *icsm*. IEEE, 40.

[7] A Apostolio and C Guerra. 1985. A fast linear space algorithm for computing longest common subsequences. (1985).

[8] Ricardo A Baeza-Yates and Chris H Perleberg. 1992. Fast and practical approximate string matching. In *Annual Symposium on Combinatorial Pattern Matching*. Springer, 185–192.

[9] Dave Binkley and Dawn Lawrie. 2015. The impact of vocabulary normalization. *Journal of Software: Evolution and Process* 27, 4 (2015), 255–273.

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the tokenisation of identifier names. In *European Conference on Object-Oriented Programming*. Springer, 130–154.

[11] Bruno Caprile and Paolo Tonella. 2000. Restructuring Program Identifier Names.. In *icsm*. 97–107.

[12] C Caprile and Paolo Tonella. 1999. Nomen est omen: Analyzing the language of function identifiers. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. IEEE, 112–122.

[13] Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda. 2015. From source code identifiers to natural language terms. *Journal of Systems and Software* 100 (2015), 117–128.

[14] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2012. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 233–242.

[15] Florian Deissenboeck and Markus Pizka. 2006. Concise and consistent naming. *Software Quality Journal* 14, 3 (2006), 261–282.

[16] Dongdong Du, Xingzhang Ren, Yupeng Wu, Jien Chen, Wei Ye, Jinan Sun, Xiangyu Xi, Qing Gao, and Shikun Zhang. 2018. Refining Traceability Links Between Vulnerability and Software Component in a Vulnerability Knowledge Graph. In *International Conference on Web Engineering*. Springer, 33–49.

[17] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. Ieee, 53–62.

[18] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. (2009).

[19] Henry Feild, David Binkley, and Dawn Lawrie. 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*.

[20] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process* 25, 6 (2013), 575–599.

[21] Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. 2012. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 103–112.

[22] Emily Hill, Zachary P Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K Vijay-Shanker. 2008. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 79–88.

[23] Yanjie Jiang, Hui Liu, Jia Qi Zhu, and Lu Zhang. 2018. Automatic and Accurate Expansion of Abbreviations in Parameters. *IEEE Transactions on Software Engineering* (2018).

[24] Danae Pla Karidi. 2016. From user graph to Topics Graph: Towards twitter followee recommendation based on knowledge graphs. In *Data Engineering Workshops (ICDEW), 2016 IEEE 32nd International Conference on*. IEEE, 121–123.

[25] Dawn Lawrie and Dave Binkley. 2011. Expanding identifiers to normalize source code vocabulary. (2011).

[26] Dawn Lawrie, Dave Binkley, and Christopher Morrell. 2010. Normalizing source code vocabulary. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 3–12.

[27] Dawn Lawrie, Henry Feild, and David Binkley. 2007. Extracting meaning from abbreviated identifiers. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*. IEEE, 213–222.

[28] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318.

[29] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[30] Nioosha Madani, Latifa Guerrouj, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2010. Recognizing words from source code identifiers using speech recognition techniques. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 68–77.

[31] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering* 34, 2 (2008), 287–300.

[32] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web* 8, 3 (2017), 489–508.

[33] Roel Popping. 2003. Knowledge graphs and network text analysis. *Social Science Information* 42, 1 (2003), 91–106.

[34] Bonita Sharif and Jonathan I Maletic. 2010. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 196–205.

[35] Mark Shtern and Vassilios Tzerpos. 2012. Clustering methodologies for software engineering. *Advances in Software Engineering* 2012 (2012), 1.

[36] Amit Singhal. 2012. Introducing the knowledge graph: things, not strings. *Official google blog* 5 (2012).

[37] Angus Stevenson. 2010. *Oxford dictionary of English*. Oxford University Press, USA.

[38] Ashish Sureka. 2012. Source code identifier splitting using yahoo image and web search engine. In *Proceedings of the First International Workshop on Software Mining*. ACM, 1–8.

[39] Lu Wang, Xiaobing Sun, Jingwei Wang, Yucong Duan, and Bin Li. 2017. Construct Bug Knowledge Graph for Bug Resolution. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 189–191.

[40] Chenyan Xiong, Russell Power, and Jamie Callan. 2017. Explicit semantic ranking for academic search via knowledge graph embedding. In *Proceedings of the 26th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 1271–1279.