Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion

Yanjie Jiang, Hui Liu, Jiahao Jin and Lu Zhang

Abstract—Although the negative impact of abbreviations in source code is well-recognized, abbreviations are common for various reasons. 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 by leveraging both semantic relation and transfer expansion. One of the key insights of the approach is that abbreviations in the name of software entity e have a 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. Another key insight is that literally identical abbreviations within the same application are likely (but not necessary) to have identical expansions, and thus the semantics-based expansion in one place may be transferred to other places. To investigate when abbreviation expansion could be transferred safely, we conduct a case study on three open-source applications. The results suggest that a significant part (75%) of expansions could be transferred among lexically identical abbreviations within the same application. However, the risk of transfer varies according to various factors, e.g., length of abbreviations, the physical distance between abbreviations, and semantic relations between abbreviations. Based on these findings, we design nine heuristics for transfer expansion and propose a learning-based approach to prioritize both transfer heuristics and semantic-based expansion heuristics. Evaluation results on nine open-source applications suggest that the proposed approach significantly improves the state of the art, improving recall from 29% to 89% and precision from 39% to 92%.

Index Terms—Abbreviation, Expansion, Transfer, Context, Quality

1 Introduction

Identifiers are common in source code. They are frequently employed to identify software entities, e.g., classes, methods, and parameters. Results of empirical studies [1], [2] suggest that identifiers account for around 70% of source code in terms of characters. Because of their great volume, the readability of such identifiers is critical for software quality, especially maintainability and readability [3], [4].

Abbreviations are common in identifiers [5], which significantly influence the readability and maintainability of software applications. An abbreviation is a sequence of characters that can be expanded into a semantically equivalent larger word or phrase [6]. To shorten identifiers, developers often employ abbreviations (e.g., fs) instead of full terms (e.g., *file systems*) to name software entities, which results in a large number of abbreviations. Results of empirical studies [7] suggest that a significant part of identifiers contain abbreviations. However, such abbreviations have severe negative impact on software engineering tasks [8], [9], [10], e.g., program comprehension [11], software maintenance [4], concept location [12], and recovery of traceability links [13], [14].

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 accuracy, researchers exploit the contexts of abbreviations, e.g., words within the same document or the same project. Exploiting such contexts has greatly improved accuracy. However, the exploited contexts are often coarse-grained and ignore semantic relations, which has a severe negative impact on the performance of abbreviation expansion [15].

To this end, in this paper, we propose a generic and accurate approach to expand identifier abbreviations. One of the key insights of the approach is that abbreviations in the name of software entity e have a 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. Another key insight is that literally identical abbreviations within the same application are likely (but not necessary) to have identical expansions, and thus the semantics-based expansion in one place may be transferred to other places. Consequently, in this paper, we propose a novel approach, called *transfer expansion* to leverage such key insight. Transfer expansion is to expand an abbrevia-

- *Y.J. Jiang, H. Liu and J.H. Jin are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: jiangyanjie@bit.edu.cn, liuhui08@bit.edu.cn, jinjiahao1993@gmail.com*
- *L. Zhang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871. E-mail: zhanglu@sei.pku.edu.cn.*
- *Corresponding Author: Hui Liu*

tion by copying the expansion of another literally identical abbreviation. To investigate when abbreviation expansion could be transferred safely, we manually analyze successful and failed transfer. Our analysis results suggest that the risk of transfer expansion varies according to various factors, e.g., length of abbreviations, the physical distance between abbreviations, and semantic relations between abbreviations. Based on these findings, we propose nine heuristics for transfer expansion.

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 both transfer heuristics and semantic-based expansion heuristics. To the best of our knowledge, it is the first learning-based approach to expanding identifier abbreviations.

The last characteristic of the proposed approach is that it leverages online resource (Wikipedia) to expand emerging abbreviations. The idea is inspired by the following observations. First, new abbreviations are emerging, and it is challenging for manually constructed abbreviation dictionaries to include such emerging abbreviations in time. Second, while encountered with unknown abbreviations in source code, developers often try to understand/expand them by Google or other Internet-based search engines. Third, Wikipedia, which is updated dynamically by numerous Internet users, contains rich information about the emerging abbreviations.

We evaluate the proposed approach on nine well-known open-source projects. The results of our k-fold evaluation suggest that the proposed approach significantly improves the state of the art. It successfully improves recall from 29% to 89% and improves precision from 39% to 92%.

The paper makes the following contributions. First, it presents a generic and accurate approach to expanding abbreviations in identifiers. The proposed approach differs from existing ones in that it exploits transfer expansion as well as 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. Second, it presents 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 paper is an expanded version of our conference paper [16] that was published recently. Compared to the conference version, this paper makes the following expansions:

- A case study to investigate the possibility and risk of transfer expansion of abbreviations.
- Significant revision on the proposed approach to leverage transfer expansion besides semantic relation that is leveraged by [16]. This paper also leverages an additional source of full terms (Wikipedia) for abbreviation expansion. Evaluation results suggest that such revisions result in significant improvement in the performance of abbreviation expansion.
- Investigation on additional research questions (RQ6, RQ7, RQ8, and RQ9).

The rest of the paper is structured as follows. Section 2 presents motivating examples, and Section 3 presents a case study. Section 4 proposes the approach whereas Section 5 presents the evaluation of the proposed approach. Section 6 discuses the limitation and implications of the proposed approach. Section 7 introduces related research, and Section 8 provides conclusions and future work.

2 Motivating Examples

In this section, we illustrate the possibility and risk of transfer expansion with examples from real-world applications. The first example (as presented in Listing 1) comes from application *Retrofit* [17]. Semantics-based expansion approaches, e.g., *kgExpander* [16], expand the abbreviation "*proto*" in "*protoAdapter_Phone*" (Line 4) successfully into "*protocol*" because the associated code comment (Line 2) contains the full term. However, such approaches fail to expand the same abbreviation "*proto*" in "*ProtoRequestBodyConverter*" (Line 8) because its contexts do not contain the full term. Other well-known approaches like *Linsen* [18] and *BigramExpander* [19] fail as well. In this case, however, we may transfer the expansion of the first "*proto*" (from *Phone.java*) to the second "*proto*" (from *ProtoRequestBodyConverter.java*). The transfer succeeds because the two abbreviations do share the same expansion.

```
1   /* Extracted from Phone.java */:
2   // Code generated by Wire protocol buffer compiler
3   public final class Phone extends Message<Phone, Phone.
         Builder>{
4   private static final class ProtoAdapter_Phone {
5          ...
6
7   /* Extracted from ProtoRequestBodyConverter.java */:
8   final class ProtoRequestBodyConverter<T>{
9          ...
10  }
```

Listing 1. Transferable Expansion

```
1   /* Extracted from SVGTextElementBridge.java */:
2   public static Rectangle2D getTextBounds
3           (BridgeContext ctx, Element elem, boolean
                checkSensitivity) {
4           ...
5       Rectangle2D ret = null;
6           ...
7   /* Extracted from AbstractGraphics2D.java */:
8   public boolean drawImage(Image img, AffineTransform xform,
9                                       ImageObserver obs){
10          boolean retVal = true;
11          if(xform.getDeterminant() != 0){
12            ...
13           retVal = drawImage(img, 0, 0, null);
14            ...
15           }
16         else{
17            ...
18           retVal = drawImage(img, 0, 0, null);
19            ...
20         }
21        return retVal;
22  }
```

Listing 2. Risky Transfer

The second example (as presented in Listing 2) comes from *Batik* [20]. Semantics-based approaches can successfully expand the abbreviation "*ret*" from variable "*ret*" (Line 5) into "*rectangle*" because the full term appears within the data type of the associated identifier. However, transferring this expansion to literally identical abbreviation "*ret*" from variable "*retVal*" (Line 10) results in incorrect expansion: the latter should be expanded into "*return*" instead of "*rectangle*".

We conclude from the examples that transfer expansion among literally identical abbreviations is feasible but risky. Consequently, before transfer expansion could be leveraged, we should figure out where transfer expansion could be applied safely.

3 Case Study

3.1 Research Questions

The case study investigates the following questions:

- **RQ1**: How often are abbreviations unique within their enclosing applications? How often do literally identical abbreviations appear in different identifiers?
- **RQ2**: How often can expansions of abbreviations be transferred safely among literally identical abbreviations?
- **RQ3**: What kind of expansions can be transferred safely, and what kind of expansions cannot be transferred?

RQ1 investigates whether (and the frequency of) abbreviations from different identifiers are literally identical. As specified in Section 1, the proposed approach is based on the assumption that literally identical abbreviations within the same application are likely (but not necessary) to share identical expansions, and thus transferring expansions among such abbreviations may improve the performance of abbreviation expansion.

RQ2 investigates another assumption of the proposed approach: literally identical abbreviations within the same application are likely to share the same expansions. The investigation not only reveals whether the assumption holds but also suggests how risky transfer expansion could be. RQ3 investigates where transfer expansion can be applied safely. The investigation may uncover factors, like the length of abbreviations and distance between abbreviations, that have a significant influence on the risk of transfer expansion.

3.2 Subject Applications and Analysis

For the case study, we select three subject applications (e.g., *DB-Manager*, *Maven*, and *Portecle*) from the applications employed by Jiang et al. [16]. Jiang et al. make the dataset publicly available on GitHub [21]. Such applications are selected because of the following reasons. First, the manual analysis conducted by Jiang et al. [16] suggests that such applications contain a large number of abbreviations. Second, some of the abbreviations have been manually expanded by Jiang et al. [16], which facilitates our manual analysis. Third, the selected applications are of different sizes and developed by different developers.

On each subject application, the manual analysis is conducted as follows. First, we collect all abbreviations (noted as L_{abb}) from different identifiers. Notably, the resulting abbreviations may be literally identical to each other, but they are taken as different abbreviations because they are from different identifiers. Second, from L_{abb} we identify unique abbreviations. Third, we remove unique abbreviations from L_{abb}, and group the remaining according to the lexicon of the abbreviations, i.e., all abbreviations from a single group are literally identical to each other. The distribution of the group size is presented in Fig. 1 (histogram). From this figure, we observe that small groups (whose size is smaller than 10) are dominating, accounting for around 70% of the groups. However, we also notice that some groups are significantly larger. For example, there are ten groups whose size is larger than one hundred. Fourth, from the resulting groups, we randomly sample 50 groups (for each subject application). For large groups that are composed of more than ten abbreviations, we randomly keep ten abbreviations for each group and discard the others. We set 10 as the maximal samples from a single group because most of the groups are smaller than 10 (as suggested in Fig. 1) and if including all abbreviations from extremely large groups could significantly bias the analysis results to the specific groups. Fifth, on each of the sampled 50 groups, the first two authors manually expand the involved abbreviations independently. 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. If it is difficult for them to reach an agreement on an abbreviation, this abbreviation is discarded to improve the confidence in manual expansion. Based on the manual expansion, we compute how often literally identical abbreviations share identical expansions and analyze when and why some of them are expanded into the same/different full terms.

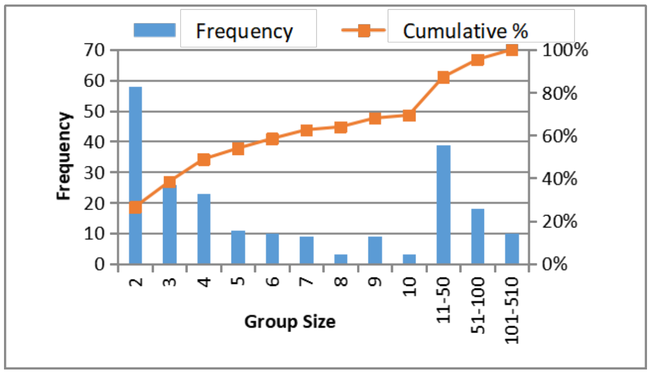


Fig. 1. Size of Non-Unique Abbreviation Groups (Histogram)

The preceding manual analysis is conducted independently on each of the subject applications. In total, we manually sample 123 groups of abbreviations that are composed of 738 abbreviations. Notably, we discard 16 abbreviations on which we make conflicting expansions, and re-sample additional abbreviations to replace them until there no enough abbreviations. When we make conflicting expansions on abbreviation ab, we discard ab only (instead of the whole group that ab belongs to), and add a new abbreviation that is literally identical to ab. We make the same expansion on most of the cases, which results in a high Cohen's Kappa coefficient of 0.76. All of the disagreements were caused by major differences (e.g., completely different words). A typical example is '*s*' that was expanded into '*Script*' and '*Security*' by different participants.

TABLE 1
Abbreviations Are Rarely Unique

Applications	Unique Abbreviations	Non-Unique Abbreviations	Ratio of Non-Unique Abbreviations	Within-File Non-Unique Abbreviations	Within-Package Non-Unique Abbreviations
DB-Manager	11	173	94%	27	83
Maven	73	2,856	98%	47	54
Portecle	91	1,862	95%	85	138
Total	175	4,891	97%	159	275

TABLE 2
Transfer Expansion Is Feasible But Risky

Applications	(PLIA) **P**airs of **L**iterally **I**dentical **A**bbreviations	Transferable PLIA	Ratio of Transferable PLIA
DB-Manager	397	306	77%
Maven	1,112	839	75%
Portecle	1,002	731	73%
Total	2,511	1,876	75%

TABLE 3
Within-File/Package PLIA

Applications	PLIA	Within-File PLIA	Within-Package PLIA
DB-Manager	965	326 (34%)	447 (46%)
Maven	234,839	2,1905 (9%)	6,103 (3%)
Portecle	93,055	11,149 (12%)	35,054 (38%)
Total	328,859	33,380 (10%)	41,604 (13%)

3.3 RQ1: Abbreviations Are Rarely Unique

The results of the analysis are presented in Table 1. Notably, Table 1 counts all abbreviations (i.e., L_{abb}) in the subject applications. The first column presents the subject applications. The second and the third column present the numbers of unique abbreviations and non-unique abbreviations, respectively. The fourth column presents the ratio of non-unique abbreviations to all abbreviations. The last two columns present the number of non-unique abbreviations that appear within the same file/package. If multiple abbreviations are literally equivalent to each other, and all of them appear in a single file only (i.e., no abbreviations in other files are literal equivalent to them), we count them as within-file non-unique abbreviations. Within-package non-unique abbreviations are counted in the same way. From this table, we make the following observations:

- First, abbreviations are rarely unique. Only 3% of the abbreviations (unique ones) are associated with a single identification. In contrast, non-unique abbreviations are dominating, accounting for 97% of the abbreviations on average.
- Second, the ratio of non-unique abbreviations to all abbreviations is high in all of the subject applications. It varies slightly from 94% to 98%, with an average of 97%.
- Third, non-unique abbreviations are often distributed across multiple files/packages. Only 159 out of the 4,891 non-unique abbreviations are in the same file (i.e., within-file non-unique abbreviations), and only 275 are in the same package.

The results are consistent with the findings reported by Newman et al. [6] in that the number of unique abbreviation-expansions is significantly larger than that of unique abbreviations because the same abbreviation may appear in different identifiers and is expanded into different expansions. According to the study by Newman et al. [6], non-unique expansions (i.e., one expansion occurs in multiple artifacts) are much more popular than unique expansions. Such findings, together with ours, reveal the distribution of unique/non-unique abbreviation and expansion.

We conclude that abbreviations are rarely unique, and non-unique abbreviations are the majority. This highlights the possibility of transfer expansion of abbreviations, i.e., leveraging the expansions of literally identical abbreviations while expanding non-unique abbreviations.

3.4 RQ2: Transfer Expansion Is Feasible But Risky

To answer RQ2, we manually expand 738 non-unique abbreviations (as specified in the last paragraph in Section 3.2) and compute how often two literally identical abbreviations share the same full expansion. The results are presented in Table 2. The first column presents the subject applications. The second column presents the number of **Pairs of Literally Identical Abbreviations** (noted as **PLIA** for short). A PLIA is noted as $< abb_i, abb_j >$ where abbreviations abb_i and abb_j appear in different identifiers but are literally identical to each other. The third column presents the number of **Transferable PLIA**. A PLIA $< abb_i, abb_j >$ is transferable if and only if abb_i and abb_j share the same full expansions. The fourth column present the ratio of transferable PLIA to all PLIA. From the table, we make the following observations:

- First, literally identical abbreviations have a great chance (varying from 73% to 77%) to share the same full expansions. Consequently transferring expansions among literally identical abbreviations is feasible.
- Second, transfer expansion is risky. It is rather often (at a chance of 25% on average) that literally identical abbreviations should be expanded into different terms. Transferring expansions on such abbreviations will result in incorrect expansions.

For a PLIA $< abb_i, abb_j >$, if abbreviations abb_i and abb_j come from the same file, we call it *within-file PLIA*. If abb_i and abb_j come from the same package, we call it *within-package PLIA*. Table 3 presents the popularity of within-file/package PLIA. It presents not only the absolute number of within-file/package PLIA but also their relative percentage (in parentheses) to the total number of PLIA. From this table, we observe that within-file PLIA accounts for only 10% of PLIA whereas within-package PLIA accounts for 13%. The results may suggest that cross-file/package transfer expansion is highly desirable.

Notably, the same abbreviation may appear in multiple PLIA. Suppose we have three literally identical abbreviations a='HR', b='HR', and c='HR'. Based on these three abbreviations, we construct three PLIA, i.e., $P_1 =< a, b >$, $P_2 =< a, c >$, and $P_3 =< b, c >$. We notice that the same abbreviation a appears in both P_1 and P_2. We also notice that abbreviations in P_1, P_2, and P_3 are literally identical to each other. If the abbreviations in one PLIA (noted uP) are literally different from abbreviations in any other PLIA, we call it (i.e., uP) unique PLIA. In our case, out of the 2,511 PLIA, 24 of them are unique PLIA.

We conclude from the preceding analysis that transfer expansion is feasible but risky. Consequently, to take full advantage of transfer expansion, we should propose automatic approaches to distinguishing cases where transfer expansion can be applied safely.

3.5 RQ3: Safe and Unsafe Transfer Expansion

To investigate where transfer expansion could be applied safely, we sample 50 *transferable PLIA* (where literally identical abbreviations share the same full expansions) and 50 *nontransferable PLIA* (where literally identical abbreviations should be expanded into different full terms). We manually analyze the resulting 100 PLIA (called *sample dataset*) to identify what kind of PLIA transfer expansion can be conducted safely. The manual analysis is conducted as follows. First, the authors manually compare transferable PLIA against nontransferable PLIA to identify some features that could be leveraged to distinguish transferable PLIA from nontransferable PLIA. The manual comparison is heuristic and empirical. After that, we statistically validate the identified features on all of the manually expanded abbreviations (i.e., 2,511 PLIA, called *whole dataset*). The manually identified potential features are the length of abbreviations, the physical distance between abbreviations, and the semantical relation between abbreviations:

- **Finding #1:** The length of abbreviations has a significant influence on the risk of transfer expansion. The shorter abbreviations are, the more risky transfer expansion is. For single-character abbreviations in the sample dataset, transfer expansion is highly risky and 84%=27/32 of the transfer expansion is incorrect. The risk is reduced to 65%=13/20 for two-character abbreviations and further reduced to 21%=10/48 for even longer abbreviations. We validate this finding on all of the manually expanded abbreviations (i.e., 2,511 PLIA). The validation results confirm that the length of abbreviations does have a significant influence on the risk of transfer expansion. The correlation coefficient between the risk of transfer and abbreviations' length is 0.55, which suggests moderate correlation.
- **Finding #2:** The physical distance between literally identical abbreviations has a significant influence on the risk of transfer expansion. On the sample dataset (i.e., 100 sample PLIA), literally identical abbreviations from the same document have a great chance of 83%=25/30 to be expanded into the same full terms. The chance is reduced to 50%=9/18 for abbreviations from the same packages and further reduced to 31%=16/52 for those from different packages. Further validation on the whole dataset (i.e., 2,511 PLIA) also confirms that physical distance does influence the risk of transfer expansion. On the whole dataset, the transfer expansion is highly secured (with a success rate of 93%) on abbreviations within the same document. The success rate is reduced significantly to 77% on abbreviations from the same packages and is further reduced to 70% on those from different packages. The success rate is significantly reduced to 46% on those from different projects. Notably, such success rates on the whole dataset are significantly higher than those reported on the sample dataset. The reason for the difference is that the whole dataset is imbalanced whereas the sampled dataset is balanced. The sample dataset is composed of 50 transferable PLIA and 50 nontransferable PLIA, and thus it is balanced. However, on the whole dataset, transferable PLIA is much more popular than nontransferable PLIA.
- **Finding #3:** Semantical relation between abbreviations may significantly reduce the risk of transfer expansion. To represent the semantical relation among software entities, we present such entities as a knowledge graph where nodes are software entities and edges are the semantic relationships between entities, e.g., inclusion (like class vs. methods), assignment (like argument vs. parameter), and inheritance (like superclass vs. subclass). On the sample dataset, the risk of transfer expansion is reduced to zero for such PLIA where literally identical abbreviations come from semantically connected identifiers (i.e., the involved identifiers are directly connected on knowledge graphs). Further validation on the whole dataset (i.e., 2,511 PLIA) suggests that transfer expansion also succeeds on all of the 34 pairs of literally identical abbreviations that are directly connected on knowledge graphs.
- **Finding #4:** Transfer expansion between one-character abbreviations from different packages is highly risky if they are not semantically connected. On the sample dataset (i.e., 100 sample PLIA), transfer expansion results in incorrect expansion for 9 out of the 10 sample *PLIA* of this category. On the whole dataset (i.e., 2,511 PLIA), the transfer expansion also fails frequently (at a chance of 85%) between one-character abbreviations from different packages.

We also apply logistic regression to the 2,511 manually expanded PLIA to validate the relationship between the dependent variable (whether a PLIA is transferable) and independent variables as a whole, i.e., the length of abbreviations, the physical distance between the abbreviations, and the semantical relation between the abbreviations (whether they are directly connected by knowledge graphs). Evaluation results suggest that R^2 of the resulting regression model is 0.229, the mean absolute error is 0.138, and the mean squared log error is 0.066. Based on the results, we conclude that the independent variables have the potential to predict the dependent variable, but making predictions with the traditional logistic regression may result in significant errors.

We conclude based on the preceding analysis that the risk of transfer expansion depends on various factors, especially the length of abbreviations, the physical distance

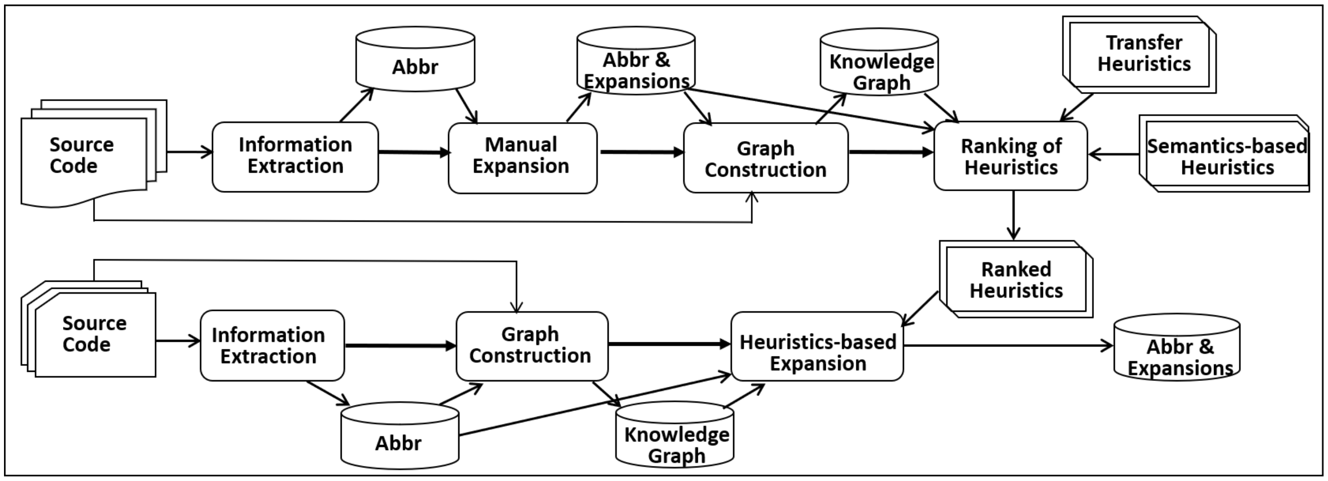


Fig. 2. Overview of the Proposed Approach

between abbreviations, and the semantical relation between abbreviations. Such findings may inspire automated approaches to identifying cases where transfer expansion could be applied. More specifically, such approaches should leverage the length of abbreviations, the physical distance between abbreviations, and the semantical relation between abbreviations. For example, according to finding #3, such automated approaches should generate high feasibility for transfer expansion if the source and target abbreviations are directly connected on their knowledge graph.

3.6 Limitations

One limitation of the case study is that it only analyzes abbreviations from three applications. Conclusions drawn on the selected applications may not hold on other applications, which may threaten the generality of the conclusions. We do not include more subject applications because the manual analysis required by the case study is time-consuming. To reduce the threat, we select applications from different domains and developed by different developers. However, including more subject applications in the future could significantly reduce the threats.

The second limitation is that the manual expansion of abbreviations could be inaccurate. The manual expansion is conducted by the authors of the paper instead of the authors of the source code. However, lacking background knowledge of the subject systems, the authors may expand the abbreviations incorrectly, which may influence the conclusions of the case study. Besides that, expansion of abbreviations is often subjective, which also results in inconsistent expansion. In the case study, the participants result in inconsistent expansion on 16 abbreviations. To improve accuracy, we exclude all such 16 abbreviations for which the authors make conflicting expansions.

The third limitation is that the manual analysis may miss some important factors/features of abbreviations that could influence the risk of transfer expansion. The uncovered features are not exclusive, and further analysis may uncover more useful features that may further improve the performance of transfer expansion. To this end, we make the analyzed data publicly available [22].

The fourth limitation is that the case study analyzes only Java code. Consequently, the conclusions of the case study may not hold on other programming languages, e.g, *C++*, and *JavaScript*.

4 Approach

4.1 Overview

An overview of the proposed approach (denoted as *tfExpander*) is presented in Fig.2. *tfExpander* is divided into two phases: the learning phase (the upper part of Fig.2) and the testing phase (the lower part of Fig.2). In the learning phase, *tfExpander* learns from a corpus of training data to prioritize a set of heuristics as follows:

- First, it extracts abbreviations from the given source code;
- Second, the resulting abbreviations are manually expanded;
- Third, for each of the identifiers containing abbreviations, it builds automatically a knowledge graph to represent identifiers that are semantically related to it.
- Fourth, based on the manual expansion and automatically constructed knowledge graphs, the proposed approach employs an algorithm to prioritize a set of heuristics so that the risk of abbreviation expansion based on the resulting heuristics could be minimized.

Output of the learning phase is a list of ranked heuristics $RL =< hs_1, hs_2 \ldots, hs_n >$. In the testing phase, *tfExpander* expands abbreviations according to the ranked heuristics as follows:

- First, it retrieves abbreviations from the given source code;
- Second, it constructs a knowledge graph for each of the resulting identifiers.
- Third, for each of the abbreviations, the proposed approach attempts to expand it by applying heuristics hs_1. If the expansion succeeds, the expansion for the abbreviation terminates. Otherwise, the proposed approach turns to the next heuristics. The iteration of the heuristics keeps going until some heuristics succeed or all of the heuristics from RL fail.

- Finally, for abbreviations where some heuristics have been applied successfully, the resulting expansions (a single expansion for each abbreviation) are presented as candidate expansions. For other abbreviations where all of the heuristics fail, no expansion is suggested.

More details of the approach are presented in the following sections.

4.2 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 abbreviation 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 are 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 a broad sense, e.g., assignment of arguments to the 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 do not influence the semantics of programs, they are rich sources 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 excludes comments within the method body.

4.3 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 direct graph:

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

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 a finer granularity than those in Section 4.2. For example, the inclusion defined in Section 4.2 is further classified into class-method, method-class, class-field, field-class, method-variable, variable-method, method-parameter, parameter-method, and variable-class.

4.4 Identification of Abbreviations

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 identifiers as follows:

1) First, we split an identifier into a sequence of hard terms according to underscores and numbers. For example, the identifier " *C2_Help* " is split into <'*C*', '*2*', and '*Help*'>.
2) Second, we split an identifier (or a hard term resulted from the preceding segmentation) into a sequence of soft words according to capital letters. Capital letters are taken as segmentation flags. An exception is the consecutive capital letters followed by lowercases (e.g., *GETName*). In this case, only the last capital letter is employed as a segmentation flag. Consequently, *"GETName"* is split into *"GET"* and *"Name"*.

The resulting soft words are taken as abbreviations if they do not appear in a generic English dictionary that is composed of full terms only [18].

4.5 Presentation of Training Data

The proposed approach is based on learning, 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 training source code, we randomly sample a number of abbreviations, and represent such abbreviations as a list:

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

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:

$$abbrExpans = \{< abbr_i, e_i, expan_i > |abbr_i \in abbrList\} \tag{5}$$

where $expan_i$ is the full expansion of $abbr_i$. Finally, we construct a knowledge graph g_i (as specified in Section 4.3) for each software entity e_i that is involved in $abbrList$. We also associate the knowledge $g(e_i)$ with the software entity e_i by a mapping $g : SE \rightarrow KG$ where SE is the involved software entities and KG are created knowledge graphs.

As a result, the training data could be represented as:

$$TrData =< abbrExpans, g > \tag{6}$$

4.6 Semantic-based Heuristics

For a given abbreviation (notated as *abbr*) within the name of an entity e_i, the proposed approach should try to find their full terms from the names of entities that are semantically related to e_i (noted as $SE(e_i)$). While Section 4.10 discusses how the proposed approach decides the order in which entities in $SE(e_i)$ should be tried, this section presents a sequence of semantic-based heuristics to find full terms of *abbr* from the name of a given entity $e_j \in SE(e_i)$.

A semantic-based heuristic could be represented as:

$$CH_i \quad = \quad < Source_j, M_k > \tag{7}$$

where $Source_j$ suggests where to search for the full terms, i.e., a node in the knowledge graph associated with the given abbreviation. M_k is a matching rule that suggests how to match the abbreviation against the given source. Such matching rules are defined as follows to handle different types of abbreviations [23], [24], [18]:

M_1 **(Acronym):** To shorten the length of an identifier that should have contained a sequence of full terms, developers often concatenate the initial characters of such terms as the abbreviation of the identifier [24]. For example, they may coin "sb" to represent "String Buffer". Based on this observation, the first matching rule M_1 searches for full terms from identifier id_j (of $Source_j$) for abbreviation *abbr* as follows. First, we segment id_j into a sequence of terms, notated as T. Second, we concatenate the initial characters of each term in T, which results in a new string noted as acr. If *abbr* is identical with acr, we return T as the expansion of *abbr*.

M_2 **(Prefix):** Developers often use a short prefix of the long term as the abbreviation. A well-known and widely used example is "str" that is frequently employed to represent "string" Based on this observation, the second matching rule M_2 searches for full terms from identifier id_j for a given abbreviation *abbr* as follows. First, we divide id_j into a sequence of terms. Second, for each term (noted as T) in the sequence, the heuristic compares T against *abbr* to test whether *abbr* is a prefix of T. If yes, T is returned as a potential expansion for *abbr*. In cases where the heuristic retrieves multiple potential expansions for the same abbreviation, it selects the shortest one.

M_3 **(Dropped-Letters):** A common way to shorten the long term is to drop some letters of the original term. For example, we may represent "index" with "idx" by dropping letters "n" and "e". Such abbreviations are often called dropped-letters. Consequently, the third matching rule M_3 searches for 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 cases where the heuristic retrieves multiple potential expansions for the same abbreviation, it selects the shortest one.

4.7 Transfer Heuristics

As suggested by the case study in Section 3, literally identical abbreviations within the same application are likely to share the same expansions. However, the case study also suggests that the risk of transfer expansion varies according to the length of abbreviations and the physical/semantic distance between abbreviations. Based on these findings, we propose a series of transfer heuristics.

The first transfer heuristic TH_1 is to expand abbreviation abb_1 by copying the full terms of another abbreviation abb_2 if abb_1 and abb_2 are literally identical and they are directly connected on the knowledge graph built for either abb_1 or abb_2. This heuristic is inspired by finding #3 of the case study.

Other transfer heuristics are presented uniformly as follows:

$$TH_i \quad = \quad < length_j, distance_k > \tag{8}$$

where $length_j$ is the length of involved abbreviations that is '1', '2', or '3+' (3 characters or even longer). $distance_k$ is the physical distance between involved abbreviations that is 0 (within the same document), 1 (from different documents of the same package), or 2 (from different packages). For example, transfer heuristic $TH_2 =< 5, 0 >$ suggests to expand abb_1 by copying the full terms of abb_2 if 1) abb_1 and abb_2 are literally identical; 2) they are defined within the same document; and 3) the length of abb_1 (and abb_2 as well) is 5 characters.

Notably, we do not have such a heuristic $TH_i =< 1, 2 >$ that would conduct transfer expansion among one-character abbreviations from different packages. We exclude this heuristic because the case study in Section 3 suggests that such kind of transfer expansion is highly risky (finding #4 of the case study).

4.8 Wiki-based Heuristics

New abbreviations are emerging, which often makes static abbreviation dictionaries obsolete. To this end, we leverage online *Wikipedia* [25] that is capturing emerging items based on open collaboration on the Internet. If we search abbreviations, e.g., *'WWW'* on Wikipedia, it often automatically jumps to the page for the full terms, e.g., *'World Wide Web'*, which may suggest that Wikipedia often contains the knowledge of abbreviations and expansions. On the Wiki pages, we also find out some clues to the relationship between abbreviations and their full terms. For example, on the page of *'World Wide Web'*, we may find such a statement *'World Wide Web* (*WWW*)'.

Based on the observations, we look for full terms for abbreviation *abbr* from Wikipedia as follows:

- First, we search for *abbr* on Wikipedia;
- Second, on the resulting page, we search for '*A* abbreviated *abbr*', '*A* abbreviation: *abbr*', '*A* shortened to *abbr*', '*abbr* shorthand for *A*', and '*abbr* acronym for *A*'. *A* is a wildcard that could be interpreted as a number of boldfaced words.
- Third, if the search in the preceding step succeeds, the longest *A* is returned as the full terms and the expansion terminates.
- Fourth, on the resulting page, we search for '*A (abbr)*' and '*A "abbr"*'. If the search in the preceding step succeeds, the longest *A* is returned as the full terms. Otherwise, we refuse to make any suggestions.

4.9 Learning Based Ranking of Heuristics

Algorithm 1 Learning Based Ranking of Heuristics

```
Input: $TrData$, //training data
       $H$ , //heuristics
Output: $RL$ //ranked heuristics
 1: $RL \leftarrow \emptyset$
 2: while $H.size() > 0$ AND $TrData.size() > 0$ do
 3:     $h \leftarrow$ SELECTONE($TrData, H$)
 4:     $RL$.append($h$)
 5:     $H$.remove($h$) //remove h from RL
 6:     //remove expanded abbreviations
 7:     $TrData$.remove($ExpandBy(h)$)
 8: end while
 9: return $RL$
10:
11: //select the best heuristic
12: function SELECTONE($TrData, H$)
13:     $L \leftarrow \emptyset$
14:     for each $h$ in $H$ do
15:         $priority \leftarrow$ PRIORITY($TrData, h$)
16:         $L$.add($h$,$priority$)
17:     end for
18:     $L.sort()$
19:     return $L.getTop()$
20: end function
```

We have a number of semantic-based heuristics (Section 4.6), a number of transfer heuristics (Section 4.7), and Wiki-based heuristics (Section 4.8). In this section, we present a learning-based algorithm (Algorithm 1) to rank such heuristics so that the most accurate and fruitful heuristics are applied first.

On the first step (Line 3 and Lines 14-17), it applies each of the heuristics from H to the whole data set $TrData$ independently, and computes their priority as follows:

$$priority(H_i) = \frac{p(H_i)}{1 + e^{-TP(H_i)}} \quad (9)$$

where $TP(H_i)$ is the number of abbreviations in the training data that are successfully expanded by heuristic H_i. $p(H_i)$ is the precision of H_i in expanding abbreviations in the training data. The algorithm selects the heuristic (noted as h) with the greatest *priority* (Lines 18-19). After that, it removes h from H (Line 5), removes all abbreviations that are expanded by h from $TrData$ (Line 7), and repeats the selection (Lines 2-8) until all of the heuristics have been selected or $TrData$ is empty. Notably, looking up abbreviation dictionaries is also taken as a potential heuristic rule. The result of the ranking is a listed of heuristics noted as $RL =< hs_1, hs_2 \ldots, hs_n >$.

4.10 Heuristics-Based Expansion

Given a list of abbreviations $abbrList$ (as defined in Equation 4) from subject applications, and a list of ranked heuristics (results of Algorithm 1 in Section 4.9), we expand such abbreviations according to Algorithm 2:

Algorithm 2 Heuristics-Based Expansion

```
Input: $abbrList$, //abbreviations to be expanded
       $RL$ //ranked heuristics
Output: $EXPs$ //expansion for $abbrList$
 1: $EXPs \leftarrow \emptyset$
 2: $TF \leftarrow \emptyset$ //Transfer heuristics
 3: while $abbrList.size() > 0$ AND $RL.size() > 0$ do
 4:     //get the top heuristic and remove it from RL
 5:     $hs_1$=RL.pop()
 6:     if isTransferHeuristics($hs_1$) then
 7:         TF.add($hs_1$)
 8:     else
 9:         for each $< abbr_i, e_i >$ in $abbrList$ do
10:             $exp_i$=Expand($abbr_i$,$e_i$,$hs_1$)
11:             if $exp_i$ != NULL then
12:                 EXPs.add($< abbr_i, e_i, exp_i >$)
13:                 $abbrList$.remove($< abbr_i, e_i >$)
14:             end if
15:         end for
16:     end if
17:     EXPs, $abbrList$=TransferExpand(EXPs, $abbrList$, TF)
18: end while
```

Algorithm 3 Transfer Expansion

```
Input: $EXPs$ , //expansion for $abbrList$
       $abbrList$, //abbreviations to be expanded
       $TF$ //transfer heuristics
Output: $EXPs$ , //expansion for $abbrList$
        $abbrList$ //abbreviations to be expanded
 1: curTF=TF
 2: while $abbrList.size() > 0$ AND $curTF.size() > 0$ do
 3:     //get the top heuristic and remove it from curTF
 4:     $tf_1$=curTF.pop()
 5:     for each $< abbr_i, e_i >$ in $abbrList$ do
 6:         $exp_i$=ExpandByTF($abbr_i$,$e_i$,$tf_1$)
 7:         if $exp_i$ != NULL then
 8:             EXPs.add($< abbr_i, e_i, exp_i >$)
 9:             $abbList$.remove($< abbr_i, e_i >$)
10:         end if
11:     end for
12: end while
```

- First, we retrieve the first heuristic hs_1 (with the highest priority) from RL (Line 5). If this is a transfer heuristic, we append it to TF (Line 7). Otherwise, we try to expand each of the abbreviations in $abbrList$ with the rule hs_1 (Lines 9-15). For a given abbreviation $< abbr_i, e_i >$ from $abbrList$ (as defined in Equation 4), we try to expand it with rule hs_1 (as defined in Equation 7). If the expansion successes (Line 11), we append the expansion to $EPXs$ (Line 12) and remove the abbreviation from $abbrList$ (Line 13) to avoid multiple expansion of the same abbreviation.
- Transfer expansion (Line 17) is activated when a new transfer heuristic is retrieved (Lines 5-7) or a semantic-based heuristic is applied thoroughly (Lines 9-15). The complete algorithm of the transfer expansion is presented in Algorithm 3.
- On the resulting abbreviation list $abbrList$ and updated heuristics RL (it drops a heuristic rule on Line 4), we

repeat the expansion until either $abbrList$ is empty or RL is empty (Line 2).

If an abbreviation results in multiple candidate expansions, we select the one with the highest frequency. That is if $< abb_i, e_i >$ is expanded by hs_1 into exp_1 for three times and into exp_2 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 have the same frequency and the same length, we randomly select one of them as candidate expansion.

5 Evaluation

In this section, we evaluate the proposed approach on nine open-source applications and compare it against the state-of-the-art approaches.

5.1 Research Questions

- **RQ4:** 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?
- **RQ5**: Does the proposed approach outperform the state-of-the-art approaches? If yes, to what extent?
- **RQ6:** How many abbreviations are expanded by transfer expansion? How accurate is the transfer expansion?
- **RQ7:** How do the Wiki-based heuristics influence the performance of the proposed approach?
- **RQ8:** How does the iterative ranking influence the performance of the proposed approach?
- **RQ9:** How do the extensions as a whole influence the performance of the proposed approach?
- **RQ10:** Does the proposed approach outperform the parameter-specific approach in expanding parameter abbreviations? If yes, to what extent and why?
- **RQ11:** How does the abbreviations' length influence the performance of abbreviation expansion?
- **RQ12:** Is the proposed approach scalable?

The proposed approach is based on the assumption that abbreviations in the name of software entity e have a great chance to find their full terms in names of software entities that are semantically related to e. Answering research question **RQ4** would validate the assumption.

Research question **RQ5** investigates the performance (e.g., precision and recall) of the proposed approach (called *tfExpander*) in comparison against the state-of-the-art approaches. To answer this question, we compare *tfExpander* against *BigramExpander* [19], and *Linsen* [18]. *BigramExpander* was proposed recently and represent the state of the art. *Linsen* is well-known and widely employed. Comparing *tfExpander* against such approaches help reveal the potential of *tfExpander*.

Research question **RQ6** investigates the usefulness of transfer expansion, i.e., how often and how well it works. Transfer expansion could be useful only if it works both frequently and accurately. Research question **RQ7** investigates the effect of Wiki-based heuristics. Notably, the prototype implementation [22] of the proposed approach leverages only the English version of Wiki [25] assuming that identifiers are written in English. Research question **RQ8** investigates the importance of iterative ranking of heuristics that is one of the key differences between the proposed approach and other heuristics-based approaches.

Research question **RQ9** investigates the cumulative impact on the performance of the extensions to the conference version (i.e., KgExpander [16]). In this paper, we make the following extensions to KgExpander: transfer expansion, Wiki-based heuristics, and iterative ranking. To answer **RQ9**, we compare the proposed approach (tfExpander) against KgExpander. Notably, KgExpander is equivalent to tfExpander if the three extensions are disabled by tfExpander.

Although parExpander proposed by Jiang et al. [15] is confined to abbreviations in parameters (actual or formal parameters), it is highly accurate [15]. Research question **RQ10** investigates whether the proposed generic approach can outperform parExpander even if only parameter abbreviations are concerned. **RQ11** concerns the impact of abbreviations' length. **RQ12** concerns the scalability of the proposed approach, i.e., whether the proposed approach can work on large projects. Notably, **RQ10-12** have been investigated in our previous study [16], and thus this paper only provides a brief summary of these results.

5.2 Subject Applications

We reuse the subject applications employed by Jiang et al. [16], [21]. Notably, three of such subject applications, i.e., *DB-Manager*, *Maven*, and *Portecle*, have been employed for the case study in Section 3. Consequently, we add three new subject applications (*FindBugs* [26], *PMD* [27], and *Subsonic* [28]) from *GitHub*.

These subject applications are employed because of the following reasons. First, they are open-source, which facilitates replication of the evaluation. Second, such applications have significant diversity. They are from different domains and developed by different programmers. They cover small projects developed by a single developer as well as large projects involving extensive cooperation among a large number of developers. They include both libraries and applications. Constructing such a diverse dataset may improve the generality of the conclusions drawn on the dataset.

5.3 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 select nine applications from GitHub based on the popularity. Notably, for such applications (e.g., *Batik*, *Bootique*, *CheckStyle*, *FileBot*, *PDFsam*, and *Retrofit*) borrowed from Jiang et al. [16]. we reuse their sampling (200 abbreviations from each of the subject applications) and their manual expansions [21]. For new applications, we randomly pick up 200 abbreviations from each of them, and manually expand the resulting abbreviations. Notably, all non-dictionary words (software terms) that are extracted automatically from identifiers are taken as potential abbreviations. However, if the non-dictionary words could not be extended into longer words/phrases, they are ignored and are excluded from the evaluation.

TABLE 4
Comparison Among Different Searching Scopes

Applications	Knowledge Graph		Enclosing Method		Enclosing File		Enclosing Project	
	$P_{presence}$	size	$P_{presence}$	size	$P_{presence}$	size	$P_{presence}$	size
Batik	84%	11	72%	25	78%	100	88 %	2146
Bootique	54%	8	53%	16	56 %	44	63%	680
CheckStyle	44%	13	43%	22	51%	97	69%	1831
FileBot	62%	9	56%	21	62%	110	75%	1566
FindBug	54%	7	50%	24	53%	72	60%	1385
PDFsam	46%	8	37%	24	49%	58	62%	998
Retrofit	25%	7	26%	20	37%	89	40%	728
PMD	52%	6	54%	28	67%	68	70%	270
Subsonic	64%	8	57%	27	63%	148	68%	780
Average	54%	9	50%	23	57%	87	66%	1154

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. If it is difficult for them to reach an agreement, they will discard such abbreviations, and resample additional abbreviations to replace them. They make the same expansion on most of the cases, which results in a high Cohen's Kappa coefficient of 0.75. To facilitate reuse and replication, we publish the resulting dataset as well as the implementation of the proposed approach [22].

The k-fold (k=9) cross-validation on the resulting dataset is conducted as follows. On each fold, a single application (one of the nine subject applications) is used as the testing subject (noted as $testingSub$) whereas the others are used as training subjects (noted as $trainingSubs$). Consequently, the validation is essentially a *leave-one-out* validation [29]. Each of the subject applications is used as the testing subject for once. On each fold of the evaluation, we evaluate the involved approaches as follows:

- **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 baseline approaches (i.e., *BigramExpander* 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.

In the evaluation, all of the involved approaches are equipped with the same abbreviation dictionaries: a generic abbreviation dictionaries [18] and a computer-specific abbreviation dictionary [30]. Employing the same dictionaries makes the comparison among the involved approaches fair, and may reduce the bias in the resulting conclusions.

5.4 RQ4: Semantic Relations are crucial for Abbreviation Expansion

To answer research question RQ4, we manually expand 1,800 abbreviations in identifiers and investigate where their full expansions could be found (The process of manual expansion is identical to Section 5.3). The key of the proposed approach is to construct a searching scope for the given abbreviations according to semantic relation. In contrast, existing approaches [11][18][23][31] 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 $P_{presence}$):

$$P_{presence} = \frac{No.\ of\ scopes\ containing\ expansions}{No.\ of\ scopes} \tag{10}$$

Greater $P_{presence}$ suggests a greater chance to find full terms from the given scope. Evaluation results of different search strategies are presented in Table 4. The first column presents subject applications. Columns 2-3 present the metrics of the proposed searching scopes, i.e., knowledge graphs. Columns 4-9 present the metrics of distance-based searching scopes. Notably, abbreviation dictionaries employed by the proposed approach (and existing approaches as well) are 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 [18]. From the table, we made the following observations:

- First, all of the searching scopes have a great chance (varying from 50% to 66%) 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 39.1%(=9/23), 10.3%(=9/87), and 0.8%(=9/1,154) of the sizes of enclosing meth-

```
1    /* Extracted from DetailASTTest.java from CheckStyle */:
2   public void testManyComments() throws Exception {
3      ...
4     try (Writer bw = Files.newBufferedWriter(file.toPath(),
          StandardCharsets.UTF_8)) {
5       bw.write("class C {\n");
6        ...
7       }
8       ...
9    }
```

Listing 3. Example of Successful Expansion

ods, 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 a great chance to find their full terms in semantic-based searching scopes (i.e., knowledge graph) as well as distance-based searching scopes. However, knowledge graphs 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.

5.5 RQ5: Improving the State of the Art

To answer research question RQ5, we compare the proposed approach (*tfExpander*) against the state-of-the-art approach (BingramExpander and Linsen). Evaluation results are presented in Table 5. The first column of Table 5 presents the subject applications. The second to the fourth columns present the results of the proposed approach, the fifth to the seventh columns present the results of *BingramExpander*. The eighth to the tenth columns present the results of *Linsen*. From the table, we made the following observations:

- First, the proposed approach is accurate. Its precision varies from 82% to 97%, with an average of 92%. High precision suggests that it makes few mistakes in abbreviation expansion, and thus it is reliable.
- Second, the proposed approach successfully expands most (89%) of abbreviations in identifiers. Such high recall indicates the high serviceability of the proposed approach.
- Third, the proposed approach significantly outperforms the state-of-the-art approaches. Compared against *BigramExpander*, it improves the precision and recall by 136%=(92%-39%)/39% and 230%=(89%-27%)/27%, respectively. Compared against *Linsen*, it improves the precision and recall by 171%=(92%-34%)/34% and 207%=(89%-29%)/29%, respectively.
- Finally, the proposed approach outperforms *Linsen* and *BigramExpander* on every subject application. For the *Linsen*, the smallest improvement in precision and recall is 41% and 43%, respectively. For the *BigramExpander*, the smallest improvement in precision and recall is 29% and 38%, respectively.

One possible reason for the improvement is that semantic relations are crucial for abbreviation expansion as suggested in Section 5.4. For example, the proposed approach successfully expands the abbreviation *"bw"* (Line 4 in Listing 3) into *"buffered writer"* based on the assignment (Line 4 in Listing 3): the left-hand and right-hand sides of the same assignment are directly connected on the knowledge graph. In contrast, *Linsen* expands it incorrectly into *"band width"* and *BigramExpander* expands it incorrectly into *"bufferedwriter writer"*. Another possible reason for the significant improvement is that the proposed approach can transfer expand some abbreviations that the state-of-the-art approaches fail to expand. For example, on abbreviation *"Ms"* in identifier *"tookMs"* from project $Retrofit$ (in *BehaviorDelegateTest.java*, Line 81), *BigramExpander* expands it incorrectly into *"millis system"* because within its searching scope (the enclosing method) it retrieves *"millis"* and *"system"*; *Linsen* expands it incorrectly into *"message"* that appears in the code comments within the same document. In contrast, the proposed approach successfully expands it into *"millisecond"* as follows. First, the proposed approach employs semantics-based heuristics to expand another *"Ms"* (in identifier *"sleepMs"* from *BehaviorCall.java*, Line 66). Second, the proposed approach transfers the expansion to the previous "Ms" (in *"tookMs"*) successfully.

It is well-known that the performance of expansion techniques may vary significantly among different abbreviation types [6]. To this end, we present the popularity of different types of abbreviations in Fig. 3, and the performance of the evaluated approaches over different abbreviation types on Table 6. From Fig. 3, we observe that acronym and prefix are the most popular abbreviation types. From Table 6, we observe that the proposed approach significantly outperforms baseline approaches on all of the abbreviation types. Notably, the proposed approach successfully expands many combination multi-word abbreviations although the heuristics proposed in Section 4.6 do not handle such combination multi-word abbreviations. Combination multi-word abbreviations are those abbreviations (excluding acronyms) that should be expanded into multi-words. The reason for the success is that Wiki-base heuristics in Section 4.8 and abbreviation dictionary based heuristics in Section 4.9 can expand combination multi-word abbreviations. For example, "regex" is expanded successfully into "regular expression" by wiki-based heuristics, and "stdout" is expanded successfully by looking up the employed abbreviation dictionary.

Notably, the proposed approach declines to make a recommendation if all of the employed heuristics fail to expand the given abbreviation. In the evaluation, it declined to make a recommendation for 71 out of the 1,800 involved abbreviations.

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.

5.6 RQ6: Transfer Expansion is Effective and Accurate

To investigate the performance of transfer expansion, i.e., how often it works and how accurate it is, we count the abbreviations expanded by transfer heuristics and compute the precision of transfer expansion. The results are presented in Table 7. The first column presents the involved subject

TABLE 5
Comparison Against State-Of-The-Art Approaches

Applications	tfExpander			BigramExpander			Linsen		
	precision	recall	F_1	precision	recall	F_1	precision	recall	F_1
Batik	95%	94%	94%	37%	27%	31%	32%	31%	31%
Bootique	82%	79%	80%	53%	41%	46%	25%	23%	24%
CheckStyle	90%	83%	86%	41%	30%	35%	49%	40%	44%
FileBot	97%	95%	96%	49%	36%	42%	28%	26%	27%
FindBug	96%	91%	93%	42%	32%	36%	39%	33%	36%
PDFsam	94%	91%	92%	35%	21%	26%	23%	21%	22%
Retrofit	89%	84%	86%	23%	12%	16%	33%	30%	31%
PMD	94%	92%	93%	38%	23%	29%	43%	35%	39%
Subsonic	92%	90%	91%	33%	20%	25%	31%	25%	28%
Average	92%	89%	90%	39%	27%	32 %	34%	29%	31%

TABLE 6
Performance Over Different Abbreviation Types

Types of Abbreviations	tfExpander			BigramExpander			Linsen		
	precision	recall	F_1	precision	recall	F_1	precision	recall	F_1
Prefix	92%	89%	90%	49%	38%	43%	43%	38%	40%
Dropped Letters	87%	83%	85%	26%	19%	22%	46%	37%	41%
Acronym	93%	90%	91%	28%	16%	20%	15%	15%	15%
Combo Multi-word	99%	94%	96%	5%	1%	2%	15%	8%	10%
Average	92%	89%	90%	39%	27%	32 %	34%	29%	31%

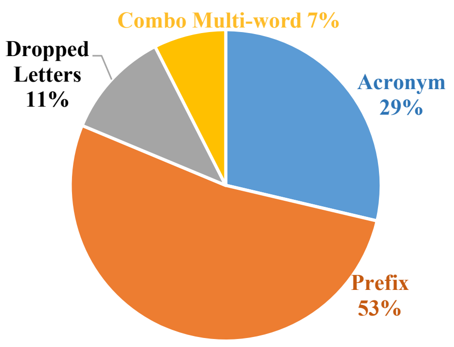


Fig. 3. Popularity of Abbreviation Types

applications. The second column presents the number of abbreviations expanded by transfer heuristics whereas the third column presents the number of correct transfer expansions. The last column presents the precision of transfer expansion. From the table, we make the following observations:

- First, transfer expansion works frequently. In total, transfer heuristics expand 510 abbreviations that account for 29.5%=510/1,729 of the abbreviations expanded by the proposed approach. Among the 1,595 correct expansions made by the proposed approach, 466 (29.2%=466/1595) are made by transfer heuristics. A typical example of the successful transfer expansion is to transfer the expansion of abbreviation *"def"* in parameter *"defSet"* (from *SVGCompositeDescriptor.java*, Line 75) to another abbreviation *"def"* in method name *"getDef"* (from *SVGCompositeDescriptor.java*, Line 48): both of them should be expanded into "definition".
- Second, transfer expansion is highly accurate. On average, its precision is 91%.
- Third, within-document transfer expansion (i.e., the source abbreviation and the target abbreviation of the transfer expansion come from the same document) accounts for only a small part of (5%=22/466) the successful transfer heuristics. Results of the case study in Section 3 suggest that within-document transfer expansion is highly accurate. However, the results in this section also suggest that focusing on within-document transfer expansion may result in a limited number (22 in the evaluation) of successful transfer expansions.

We notice that the precision of transfer expansion reduces significantly on subject application *Bootique*. The proposed approach expands abbreviation *"md"* (from *ModuleMetadataIT.java*, line 24) into *"modules"* with semantics-based heuristics. However, the expansion is incorrect and the correct expansion should be *"metadata"*. This incorrect expansion is then transferred to the other 8 literally identical abbreviations, which results in 8 incorrect transfer expansion although the root cause of the incorrect expansion is semantics-based heuristics. If the semantics-based expansion on the first *"md"* succeeds, however, the precision of transfer expansion would have significantly increased to 91% on *Bootique*. The analysis results may suggest that accurate transfer expansion depends on accurate expansion made by other heuristics. For example, disabling the iterative ranking of heuristics results in the worse ranking, and thus more abbreviations are expanded incorrectly (as shown in Section 5.8). As a result of the increased incorrect expansion, the precision of transfer expansion is reduced significantly from 92% to 81%.

We conclude that transfer expansion is effective and accurate. However, the high performance of transfer expansion depends on accurate expansion made by other heuristics.

TABLE 7
Performance of Transfer Expansion

Applications	Transfer Expansions	Correct Transfer Expansions	Precision
Batik	40	38	95%
Bootique	45	33	73%
CheckStyle	68	59	87%
FileBot	44	44	100%
FindBug	53	48	91%
Pdfsam	45	42	93%
Retrofit	83	76	92%
PMD	63	61	97%
Subsonic	69	65	94%
Total	510	466	91%

TABLE 8
Influence of Wiki-Based Heuristic

Applications	Default Setting		Disabling Wiki-Based Heu	
	precision	recall	precision	recall
Batik	95%	94%	95%	93%
Bootique	82%	79%	82%	76%
CheckStyle	90%	83%	90%	79%
FileBot	97%	95%	97%	93%
FindBug	96%	91%	97%	87%
PDFsam	94%	91%	94%	88%
Retrofit	89%	84%	88%	75%
PMD	94%	92%	95%	88%
Subsonic	92%	90%	92%	82%
Average	92%	89%	92%	84%

5.7 RQ7: Influence of Wiki-based Heuristics

To answer RQ7, we disable Wiki-based heuristics and repeat the evaluation. Evaluation results are presented on Table 8. The second and third columns present the performance of the proposed approach with the default setting. The last two columns present the performance of the approach where Wiki-based heuristics are disabled. From this table, we make the following observations:

- First, disabling Wiki-based heuristics results in a significant reduction in the recall. It is reduced from 89% to 84% whereas the average precision keeps stable. The results may suggest that Wiki-based heuristics can expand a number of abbreviations that cannot be expanded by other heuristics.
- Second, the effect of Wiki-based heuristics varies from project to project. Disabling Wiki-based heuristics results in a smaller reduction in recall on applications where other heuristics work well. For example, on subject applications $Batik$ and $FileBot$, the proposed approach (without the Wiki-based heuristics) successfully expands most (93%) of the abbreviations. As a result, few of the sampled abbreviations are left to Wiki-based heuristics, which results in the smallest effect of the heuristics. In contrast, on projects (e.g.,*Retrofit*) where other heuristics fail frequently, Wiki-based heuristics result in greater improvement in performance. The results suggest that Wiki-based heuristics are complementary to semantics-based heuristics and transfer heuristics.

We manually analyze the abbreviations expanded successfully by Wiki-based heuristics. Results suggest that most of such abbreviations are confined to a small field. For example, "*yaml*", whole full terms are "*YAML Ain't Markup Language*", refers to a very special human-readable data-serialization language. Such abbreviations are not common, and thus they have little chance to get into generic abbreviation dictionaries. The Wiki-based heuristics also successfully expand some abbreviations (e.g., "*bom*") that become popular after the employed abbreviation dictionaries were created.

We conclude based on the preceding analysis that the Wiki-based heuristics are effective and accurate.

TABLE 9
Influence of Iterative Ranking

Applications	Iterative Ranking		Flat Ranking	
	precision	recall	precision	recall
Batik	95%	94%	93%	92%
Bootique	82%	79%	78%	75%
CheckStyle	90%	83%	83%	75%
FileBot	97%	95%	91%	89%
FindBug	96%	91%	90%	83%
PDFsam	94%	91%	86%	82%
Retrofit	89%	84%	83%	72%
PMD	94%	92%	86%	81%
Subsonic	92%	90%	89%	80%
Average	92%	89%	87%	81%

5.8 RQ8: Influence of Iterative Ranking

To answer RQ8, we disable the iterative ranking of heuristics and repeat the evaluation. One of the key differences between the proposed approach and existing ones (e.g., *kgExpander* [16]) is that the proposed approach employs iterative ranking whereas existing approaches employ flat ranking. The flat ranking is to apply each of the heuristics independently to the same dataset once and for all, compute their priorities, and sort such heuristics according to the resulting priorities. A key advantage of iterative ranking is that it considers the correlation among the heuristics whereas flat ranking does not. Evaluation results are presented in Table 9. From this table, we make the following observations:

- First, replacing iterative ranking with flat ranking results in a significant reduction in both precision and recall. It reduces precision from 92% to 87%, and recall from 89% to 81%.
- Second, we observe that the precision of transfer expansion is reduced significantly from 92% to 87% when iterative ranking is replaced with flat ranking. One possible reason is that iterative ranking results in the more frequent application of more accurate heuristics, which in turn guarantees the quality of the source (expansions) of transfer expansion.

We conclude that iterative ranking of heuristics is critical for the high performance of the proposed approach. It is also critical for the performance of transfer expansion.

5.9 RQ9: Cumulative Impact of Extensions

To investigate the cumulative effect of the extensions made to the conference version (i.e., KgExpander), we compare the proposed approach against KgExpander. Evaluation results

TABLE 10
Cumulative Impact of Extensions

Applications	tfExpander		KgExpander	
	precision	recall	precision	recall
Batik	95%	94%	90%	86%
Bootique	82%	79%	81%	74%
CheckStyle	90%	83%	82%	71%
FileBot	97%	95%	91%	86%
FindBug	96%	91%	93%	82%
PDFsam	94%	91%	75%	70%
Retrofit	89%	84%	79%	64%
PMD	94%	92%	84%	75%
Subsonic	92%	90%	91%	77%
Average	92%	89%	85%	76%

TABLE 11
Expansions of Parameter Abbreviations

Applications	tfExpander			parExpander		
	precision	recall	F_1	precision	recall	F_1
Batik	96%	96%	96%	96%	50%	66%
Bootique	77%	74%	76%	95%	53%	68%
CheckStyle	89%	83%	86%	100%	93%	96%
FileBot	96%	95%	95%	97%	67%	79%
FindBug	98%	91%	95%	92%	68%	78%
PDFsam	97%	94%	96%	89%	59%	71%
Retrofit	87%	81%	84%	100%	27%	43%
PMD	98%	92%	95%	92%	72%	81%
Subsonic	90%	82%	86%	91%	64%	75%
Average	93%	89%	91%	94%	62%	75 %

are presented in Table 10. From this table, we observe that the extensions significantly improve the performance of the proposed approach: They improve the precision from 85% to 92% and recall from 76% to 89%. The improvement in recall is up to 13 percentage points. We also observe from Table 10 that the extensions result in significant performance improvement on each of the subject applications.

5.10 RQ10: Expansion of Parameter Abbreviations

Although parExpander proposed by Jiang et al. [15] is confined to parameters, 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.

We compare the proposed approach against parExpander on the 442 parameter abbreviation that parExpander was intentionally designed to expand. Results are presented in Table 11. From this table, we observe that parExpander is highly accurate in expanding parameter abbreviations, which is inconsistent with previous study [15]. We also observe that the proposed approach outperforms parExpander. It improves the average F1 score from 75% to 91% and improves the average recall significantly from 62% to 89% whereas its precision (93%) is comparable to that (94%) of parExpander even if only parameter abbreviations are concerned.

Notably, only 25%(=442/1,800) 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.

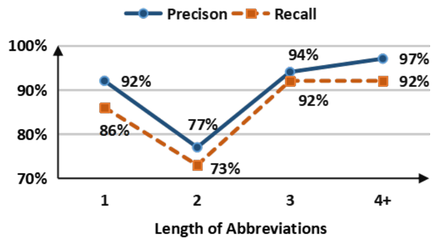


Fig. 4. Impact of Abbreviations' Length

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

5.11 RQ11: Impact of Abbreviations' Length

Fig. 4 presents the impact of abbreviations' length on the performance of abbreviation expansion. From Fig. 4, we observe that the proposed approach works well for short abbreviations, especially single-character abbreviations. We also observe that the proposed approach fails to expand a significant part (27%) of the two-character abbreviation. We manually analyze the failed cases to figure out the major reason for the failure. We find that 60% 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".

5.12 RQ12: Scalability

We investigate the scalability of the proposed approach by analyzing the relationship between the 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 Windows7. Notably, the execution time includes training time (as specified in Section 4.9). Based on the results, we make the following observations:

- First, the execution time increases when the size of projects increase;
- Second, the proposed approach is efficient, taking around 16 minutes only to parse and expand 4,701 abbreviations in the largest application (CheckStyle) that contains more than 38 thousand of lines of source code.
- Third, compared to the conference version (kgExpander), tfExpander is more time-consuming. For example, on the largest application (CheckStyle), the execution time is increased significantly from 5 minutes (kgExpander) to 16 minutes (tfExpander).

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

5.13 Threats to Validity

A threat to construct validity is that the dataset involved in the evaluation may be inaccurate because the involved abbreviations (from the three newly added applications) are manually expanded by three students instead of the original developers. To reduce the threat, we request the participants to work together and exclude expansions where they cannot reach an agreement. We also make the evaluation data publicly available on GitHub [22].

Another threat to construct validity is that the manual expansion is subjective. If the participants know in advance how the proposed approach works, they may expand the abbreviations as the proposed approach does. As a result, the resulting dataset may contain serious bias for the proposed approach. To reduce the threat, we recruit students who are not aware of the proposed approach.

A threat to external validity is that the evaluation is conducted on a limited number of applications because manual expansion is time-consuming. Special characteristics of such projects may bias the conclusions. To reduce the threat, we select projects from different developers, and evaluation results suggest that the proposed approach improves the state of the art on each of the projects.

6 Discussion

6.1 Relationship among Abbreviations

Our case study in Section 3 and our evaluation in Section 5 suggest that the relationship among abbreviations are useful clues for abbreviation expansion. More specifically, we explore and leverage both lexical similarity among abbreviations (i.e., whether they are literally identical) and semantic relation among abbreviations (i.e., whether they are directly connected on their knowledge graphs). Evaluation results suggest that leveraging such relationship can significantly improve the performance of abbreviation expansion.

Our initial investigation on the relationship among abbreviations may inspire further research in this area. Additional relationships among abbreviations deserve further investigation, and even the explored relationship (i.e., the lexical similarity and semantical relationship) could be explored further. For example, for lexical similarity among abbreviations, we only measure whether two abbreviations are literarily identical. However, fine-grained lexical relation (e.g., whether one is a prefix or subsequence of the other) and quantitative measurements (i.e., how similar they are) have not yet been explored.

6.2 Online Documents for Abbreviation Expansion

In this paper, we leverage Wikipedia as the source of full terms. To the best of our knowledge, we are the first to leverage such online and dynamic data for abbreviation expansion. Different from manually constructed abbreviation dictionaries, Wikipedia has a greater chance to include emerging and less popular abbreviations (and their full terms).

Our investigation may inspire further research in exploring new sources of full terms. Frequently explored sources by existing approaches include source code (including code comments), documents of applications, documents of programming languages, and dictionaries (both generic English dictionaries and abbreviation dictionaries). However, our work suggests that more data sources could be leveraged for abbreviation expansion. New sources like test cases, bug reports, and technique forums (like Stack Overflow) may deserve further research.

7 Related Work

7.1 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 [32]. An abbreviation dictionary contains a list of well-known abbreviations and their corresponding full terms. Abbreviation expansion based on abbreviation dictionaries is often highly accurate. However, such abbreviation dictionaries are often constructed manually, which significantly limits the size of such dictionaries [33]. As a result, approaches that are completely based on such dictionaries can expand only a small number of abbreviations, resulting in a low recall.

Generic English dictionaries are also employed for abbreviation expansion [34]. 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 cases, 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 the generic English dictionary, resulting in a great challenge in selecting the correct one. To this end, more advanced approaches look for full terms from the context of abbreviations [35][36]. Such contexts include comments [37], and enclosing methods/documents/projects [31], [18]. Corazza et al. [18], Lawrie et al. [37] and Madani et al. [35] suggest looking for full terms from comments of source code. However, developers rarely write comments. Lawrie et al. [31] suggest searching the enclosing methods whereas Hill et al. [23] propose a more complex approach to search enclosing methods, enclosing classes, and enclosing projects in order. Abdulrahman Alatawi et al. [38] leverage the Bayesian unigram-based inference model to find full terms from source code. To the best of our knowledge, Abdulrahman Alatawi et al. [19] 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 a severe negative impact on the performance of abbreviation expansion.

Besides the generic approaches introduced in the preceding paragraphs, Jiang et al. [15] 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. [39] 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. [36] 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. [23] and Carvalho et al. [11] sort such potential expansion according to their frequency, and recommend the top one. Lawrie et al. [31] choose the one who has the highest lexical similarity with the given abbreviation. Guerrouj et al. [40] and Corazza et al. [18] employ graph-based matching algorithms. The 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 the Dijkstra algorithm. The one with the shortest path is recommended as the most likely expansion. Jiang et al. [15] propose a series of heuristics to search for potential expansions. In cases where multiple potential expansions exist, they select the shortest one.

Newman et al. [6] conduct an empirical study of abbreviations. They analyze 861 abbreviation-expansion pairs extracted from five open-source applications, and some interesting results are reported. One of the interesting findings of their study is that language document, project document, and source code contain similar numbers of expansions whereas language document is the primary source of unique expansions (that appear in a singly data source only). Another finding of the study is that full terms of a single abbreviation (e.g., '*information*' and '*retrieval*' for abbreviation '*IR*') may not appear adjacently in the data source. They also present insightful suggestions on how to report effectiveness on different abbreviations types. They find that most papers report overall performance only whereas only around half of them report performance on different abbreviation types. To the best of our knowledge, this is the first large-scale empirical study on abbreviations.

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.

7.2 Segmentation of Identifier Names

The segmentation of identifiers is to decompose identifiers into sequences of soft words [7]. 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 example, we can segment identifiers according to the positions of some characters, such as underscores, and capital characters [41]. 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. [42] 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. [43] 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. [36] proposed an approach named $GenTest$, and Carvalho et al. [11] 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. [44] proposed an approach named $TIDIER$, and an enhanced version named $TRIS$ [40]. $TIDIER$ 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. [18] employ an approximate string matching techniques (called Baeaz-Yates and Perleberg) [45] to segment identifiers. Ashish Sureka [46] 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.

7.3 Knowledge Graph

A knowledge graph is a graph presenting entities and their semantical relations. Vertexes of knowledge graphs represent entities whereas edges present relations between the entities [47]. We may fuse information from a variety of data sources to construct a comprehensive semantic network [48]. 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 [49] and search engines [50].

In the field of software engineering, knowledge graphs have been successfully employed to facilitate bug resolution [51], to refine traceability links [52], and to improve API caveats accessibility [53]. Wang et al. [51] construct a bug knowledge graph based on their self-constructed bug knowledge. In the bug knowledge graph, entities are descriptions of bugs, commits, 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 about a software bug issue. Du et al.

[52] 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. [53] construct an API caveat knowledge graph. They extract API caveat sentences from API documents and link these sentences to API entities to form this API caveat knowledge graph. It improves the API caveat accessibility, and developers can avoid many unexpected programming errors. All such successful applications of knowledge graphs suggest that the knowledge graph has great potential in resolving software engineering tasks. This is one of the reasons why the proposed approach employs knowledge graphs.

8 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 as well as transfer expansion. One of the key insights 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. Another key insight of the approach is that literally identical abbreviations within the same application are likely to have identical expansions, and thus the semantics-based expansion in one place may be transferred to other places. We explore the possibility of transfer expansion of abbreviations in source code and manually identify the major features of abbreviations that may influence the risk of transfer expansion.

Based on the preceding key insights, we propose an automatic approach, called *tfExpander*, to expanding abbreviations in identifiers by leveraging both semantic relations among software entities and transfer expansion. Besides semantic-based heuristics that leveraging semantic relations among identifiers, we also propose a sequence of heuristics to transfer expansion among literally identical abbreviations, and heuristics to expand abbreviations according to online Wikipedia. Finally, we prioritize all such heuristics with a learning algorithm on manually created training data. We evaluate the proposed approach on real-world applications. The results of the leave-one-out cross-validation suggest that the proposed approach significantly improves the state of the art in abbreviation expansion: improving recall from 29% to 89% and precision from 39% to 92%.

In the future, it would be interesting to evaluate the proposed approach on bigger data, involving more subject applications and more abbreviations. Large scale evaluation in the future may reduce the threats to validity. The proposed approach also has the potential to be extended in the future to programming languages other than Java. Its prototype implementation is confined to Java. However, the key rationale of the proposed approach is not confined to Java. In the future, it is also interesting to investigate cross-project transfer expansion, although the case study in Section 3 suggests that cross-project transfer expansion often (at a chance of 54%) results in incorrect expansion. It could be valuable to uncover special cases where cross-project transfer expansion works.

As suggested by an existing study [6], expansion words of a single abbreviation may be non-adjacent in the source of expansions (e.g., documents). However, the proposed approach, in the current form, could not handle such cases. In the future, it could be interesting to extend the proposed approach to handle such cases.

Acknowledgment

The authors would like to thank the anonymous reviewers from ESEC/FSE2019 for their insightful comments and constructive suggestions.

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

References

[1] F. Deissenboeck and M. Pizka, “Concise and consistent naming,” *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.
[2] F. Deissenboeck, M. Pizka, and T. Seifert, “Tool support for continuous quality assessment,” in *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*. IEEE, 2005, pp. 127–136.
[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 281–293.
[4] D. Binkley and D. Lawrie, “The impact of vocabulary normalization,” *Journal of Software: Evolution and Process*, vol. 27, no. 4, pp. 255–273, 2015.
[5] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “Effective identifier names for comprehension and memory,” *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007.
[6] C. D. Newman, M. J. Decker, R. Alsuhaibani, A. Peruma, D. Kaushik, and E. Hill, “An empirical study of abbreviations and expansions in software artifacts,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 269–279.
[7] C. Caprile and P. Tonella, “Nomen est omen: Analyzing the language of function identifiers,” in *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE, 1999, pp. 112–122.
[8] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE transactions on software engineering*, vol. 28, no. 10, pp. 970–983, 2002.
[9] J. I. Maletic and A. Marcus, “Supporting program comprehension using semantic and structural information,” in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 103–112.
[10] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, “Repairing programs with semantic code search (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.
[11] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, “From source code identifiers to natural language terms,” *Journal of Systems and Software*, vol. 100, pp. 117–128, 2015.
[12] A. Marcus, D. Poshyvanyk, and R. Ferenc, “Using the conceptual cohesion of classes for fault prediction in object-oriented systems,” *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.
[13] D. Lucia *et al.*, “Information retrieval models for recovering traceability links between code and documentation,” in *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 2000, pp. 40–49.
[14] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc, “Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis,” in *2008 16th IEEE International Conference on Program Comprehension*. Ieee, 2008, pp. 53–62.
[15] Y. Jiang, H. Liu, J. Q. Zhu, and L. Zhang, “Automatic and accurate expansion of abbreviations in parameters,” *IEEE Transactions on Software Engineering (Early Access), DOI: 10.1109/TSE.2018.2868762*, pp. 1–15, 2018.

[16] Y. Jiang, H. Liu, and L. Zhang, "Semantic relation based expansion of abbreviations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 131–141.

[17] *Retrofit*, https://github.com/square/retrofit, 2019.

[18] A. Corazza, S. Di Martino, and V. Maggio, "Linsen: An efficient approach to split identifiers and expand abbreviations," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 233–242.

[19] A. Alatawi, W. Xu, and J. Yan, "The expansion of source code abbreviations using a language model," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 370–375.

[20] *Maven*, https://github.com/apache/xmlgraphics-batik, 2020.

[21] Y. J. et al., "Replication Package of kgExpander," https://github.com/4DataShare/AbbExpansion, 2019.

[22] Prototype Imeplementation and Evaluation Data, "Transfer expansion of abbreviation," https://github.com/DataPublic/TransferExpander, 2019.

[23] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker, "Amap: automatically mining abbreviation expansions in programs to enhance software maintenance tools," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 79–88.

[24] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 130–154.

[25] *Wikipedia*, https://en.wikipedia.org/wiki/Wikipedia, 2019.

[26] *FindBugs*, https://github.com/findbugsproject/findbugs, 2019.

[27] *PMD*, https://github.com/pmd, 2019.

[28] *Subsonic*, https://github.com/daneren2005/Subsonic, 2019.

[29] M. Kearns and D. Ron, "Algorithmic stability and sanity-check bounds for leave-one-out cross-validation," *Neural computation*, vol. 11, no. 6, pp. 1427–1453, 1999.

[30] *Dictionary*, https://www.webpages.uidaho.edu/ jory/babel.html, 2007.

[31] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 113–122.

[32] E. Adar, "Sarad: A simple and robust abbreviation dictionary," *Bioinformatics*, vol. 20, no. 4, pp. 527–533, 2004.

[33] B. Caprile and P. Tonella, "Restructuring program identifier names." in *icsm*, 2000, pp. 97–107.

[34] A. Stevenson, *Oxford dictionary of English*. Oxford University Press, USA, 2010.

[35] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 2010, pp. 68–77.

[36] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 3–12.

[37] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 2007, pp. 213–222.

[38] A. Alatawi, W. Xu, and D. Xu, "Bayesian unigram-based inference for expanding abbreviations in source code," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 543–550.

[39] A. Apostolio and C. Guerra, "A fast linear space algorithm for computing longest common subsequences," 1985.

[40] L. Guerrouj, P. Galinier, Y.-G. Guéhéneuc, G. Antoniol, and M. Di Penta, "Tris: A fast and accurate identifiers splitting and expansion algorithm," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 103–112.

[41] B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and under_score identifier styles," in *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 2010, pp. 196–205.

[42] H. Feild, D. Binkley, and D. Lawrie, "An empirical comparison of techniques for extracting concept abbreviations from identifiers," in *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA'06)*, 2006.

[43] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.

[44] L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "Tidier: an identifier splitting approach using speech recognition techniques," *Journal of Software: Evolution and Process*, vol. 25, no. 6, pp. 575–599, 2013.

[45] R. A. Baeza-Yates and C. H. Perleberg, "Fast and practical approximate string matching," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 1992, pp. 185–192.

[46] A. Sureka, "Source code identifier splitting using yahoo image and web search engine," in *Proceedings of the First International Workshop on Software Mining*. ACM, 2012, pp. 1–8.

[47] R. Popping, "Knowledge graphs and network text analysis," *Social Science Information*, vol. 42, no. 1, pp. 91–106, 2003.

[48] H. Paulheim, "Knowledge graph refinement: A survey of approaches and evaluation methods," *Semantic web*, vol. 8, no. 3, pp. 489–508, 2017.

[49] D. P. Karidi, "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, 2016, pp. 121–123.

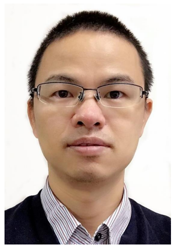
[50] C. Xiong, R. Power, and J. Callan, "Explicit semantic ranking for academic search via knowledge graph embedding," in *Proceedings of the 26th international conference on world wide web*. International World Wide Web Conferences Steering Committee, 2017, pp. 1271–1279.

[51] L. Wang, X. Sun, J. Wang, Y. Duan, and B. Li, "Construct bug knowledge graph for bug resolution," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 189–191.

[52] D. Du, X. Ren, Y. Wu, J. Chen, W. Ye, J. Sun, X. Xi, Q. Gao, and S. Zhang, "Refining traceability links between vulnerability and software component in a vulnerability knowledge graph," in *International Conference on Web Engineering*. Springer, 2018, pp. 33–49.

[53] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 183–193.

Yanjie Jiang received her BE degree from the College of Information Engineering, Northwest A&F University in 2017. She is currently working toward a Ph.D. degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. Her current research interests include software refactoring and software quality.



Hui Liu is a professor at the School of Computer Science and Technology, Beijing Institute of Technology, China. He received a BS degree in control science from Shandong University in 2001, an MS degree in computer science from Shanghai University in 2004, and a Ph.D. degree in computer science from Peking University in 2008. He was a visiting research fellow in centre for research on evolution, search, and testing (CREST) at University College London, UK. He served on the program committees and organizing committees of prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC. He is particularly interested in software refactoring, AI-based software engineering, and software quality. He is also interested in developing practical tools to assist software engineers.

Jiahao Jin received BS degree from Gengdan Institute of Beijing University of Technology in computer science in 2017. He is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. He is particularly interested in software refactoring and software evolution.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both Ph.D. and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification, and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.