2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00115

A Cross-Language Name Binding Recognition and Discrimination Approach for Identifiers

Yue Ju, Yixuan Tang, Jinpeng Lan, Xiangbo Mi, and Jingxuan Zhang
College of Computer Science and Technology
Nanjing University of Aeronautics and Astronautics
Nanjing, China
{juyue, tangyixuan, lanjinpeng, mixiangbo, jxzhang}@nuaa.edu.cn

***Abstract*—Software developers usually rename identifiers and propagate the renaming based on the name binding of identifiers. Currently, software applications are usually developed using more than one language to enhance their functions and behaviors. Hence, when an identifier renaming is performed, it frequently affects more than one language in the multiple-language software applications. However, existing name binding approaches for identifiers only focus on a specific single language without considering the cross-language scenario. In this paper, we propose a cross-language name binding approach for the Java framework based on the deep learning model. Specifically, we first detect the potential name binding pairs via string matching. By analyzing the name binding pairs, the context, and the framework rules of identifiers, we extract several deep semantic features of identifiers and employ the BERT pre-trained model to recognize the name binding for unique identifiers, and further combine several classifiers to discriminate the name binding for duplicate identifiers. Our approach is evaluated on a manually constructed experimental dataset from 10 multiple-language projects. Experimental results demonstrate that our approach can achieve the average F-Measure of 85.14% in unique identifiers and 86.57% in duplicate identifiers, which significantly outperforms the baseline approaches. We also compare the performance of our approach against IntelliJ IDEA to further show its usefulness for developers in the real scenario.**

***Index Terms*—Cross Language, Name Binding, Deep Learning**

I. Introduction

Identifiers take up almost 70% of source code lexicon [1], and meaningful identifiers are the most crucial information for developers to understand the lexical and semantic meanings of the programs [2]. Along with the development and evolution of software, the lexical meaning of identifiers may no longer correctly reflect their functions or behaviors, which requires the renaming of identifiers [3]. Before performing the identifier renaming, developers should employ or design an name binding approach to make sure that all the identifiers in different languages are linked together in the same project [4]. Hence, once an identifier in the source code needs to be renamed, we can easily find out other binded identifiers in the same or other languages, which need also to be renamed to perform the renaming propagation. In such a way, we can guarantee that the programs can be still compiled after the renaming of specific identifiers and the propagation of the renaming.

Jingxuan Zhang is the corresponding author.

Nowadays, Multiple-Language Software Applications (MLSAs) utilizing various frameworks and languages are frequently developed [5], making up the majority of the newly-created software projects. Within MLSAs, if we want to rename the identifiers of one language, we need to also rename all the binded identifiers in other languages to ensure the correction of renaming propagation. All involved languages must undergo the identifier renaming operations to guarantee the project behaves correctly afterward [6]. Hence, identifier name binding is crucial for MLSAs.

In the literature, researchers have proposed several name binding approaches for identifiers. However, these approaches only focus on the same single language, i.e., within Java itself [7], without considering the name binding among multiple languages, such as among Java, XML, and JSON. In addition, even the most extensively used Integrated Development Environments (IDEs) [8] cannot fully and accurately bind identifiers across different languages in the same project. These IDEs only search for the identifiers with the same name to perform the name binding and propagate the renaming across different languages. However, only relying on the name lexical matching could bind a lot of unrelated identifiers and create a lot of bugs when performing the renaming propagation.

We have discovered that cross-language name binding is typically built between the generic (e.g., Java) and the domain-specific (e.g., XML) languages through studying the framework fundamentals and analyzing a large scale of real-world projects. String matching may help to determine the cross-language name binding for classes and methods. Due to the rule definitions of frameworks, it is difficult to identify the cross-language name bindings for fields. In such a situation, developers must search for these fields in several languages in the same project. It is challenging for developers to determine whether the name binding holds, because fields may repeatedly appear multiple times and require precise distinction.

Hence, this study focuses on proposing a cross-language name binding approach for fields to achieve renaming recognition and discrimination during the software renaming execution and propagation, so programs can still be compiled and run after the identifier binding and renaming. Specifically, we first construct name binding pairs among the Java language and other languages with the same name using string matching. At the same time, we also extract the programming context

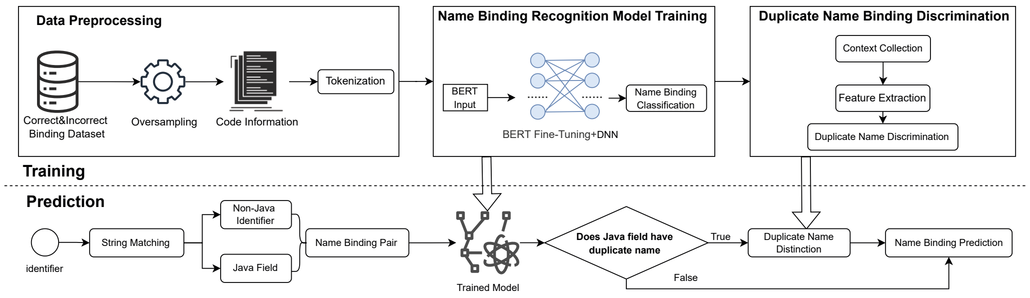

Fig. 1: The workflow of the proposed cross-language name binding approach.

information of field names. Then, the name binding pairs as well as their programming context are mined by the BERT fine-tuning model to obtain the deep semantic information [9]. Next, by analyzing the programming context information and framework rules, we can train a name binding recognition classifier. If a field has duplicate names in different classes, we further utilize the assemble learning to combine different classifiers to discriminate the exactly correct names that need to be binded for this field.

We manually construct a benchmark dataset containing 10 open source projects programmed by Java frameworks with multiple languages. Based on this dataset, we evaluate the performance of our approach. Experimental results show that our approach achieves an average F-Measure of 85.14% in single unique fields and 86.57% in duplicate fields. In addition, our approach can also significantly outperform the existing baseline approaches by up to 57.41% and 71.09% in terms of the average F-Measure.

In summary, the primary contributions of this paper are listed as follows.

- In order to achieve cross-language name binding recognition and discrimination, we propose a new approach that extracts the semantic information of identifiers using the deep learning model and integrates the programming context as well as framework basic rules.
- We conduct extensive experiments to demonstrate the effectiveness of our approach in the cross-language name binding.
- To advance the study of cross-language name binding, we manually construct a dataset containing ten projects with cross-language name bindings and make it as well as the replication package available to the public[1].

This paper is structured as follows. In Section II, we present the details about our approach. Our experimental design and results are illustrated in Sections III and IV, respectively. Threats to validity are shown in Section V and the related work are presented in Section VI. Finally, this work is concluded in Section VII.

II. FRAMEWORK

We propose a cross-language name binding approach for the Java frameworks by thoroughly analyzing the source code, programming context, and fundamental principles of the frameworks. Fig. 1 illustrates the overall framework of our approach, which typically follows the supervised work-flow with training and prediction. This approach contains four main components, including data preprocessing, name binding recognition model training, duplicate name binding discrimination, and cross-language name binding prediction. We present the details of these components as follows.

A. Data Preprocessing

To accurately bind identifiers in multiple languages, our approach primarily employs the BERT fine-tuning model [9] to capture deep semantic information of identifiers and determine the name binding pairs. To use the BERT model, we shall first extract and process the original data so that they are suitable for the input of the BERT model. Specifically, we first employ the string matching to detect the potential name bindings, since if two identifiers are binded, they should have the same name. We calculate the lexical similarity between each pair of identifiers. If two identifiers are exactly the same, they are binded. Through this step, we can obtain a series of name binding pairs for identifiers.

In practice, developers may define a lot of identifiers with the same name but in different files. Hence, the string matching step will result in that the number of correct name binding pairs is far less than that of incorrect name binding pairs. To reduce such an impact, we employ an oversampling approach[2] by simply copying the correct name binding pairs to enhance their weights.

Each pair of name binding contains non-alphabetic and numeric characters, which are removed. All the remaining tokens are kept and converted to text vectors, and separated into lists maintaining the order of the tokens. After that, the token sequence of each name binding pair is made up of three parts, including the file name, the identifier name, and the

[1]https://github.com/sharonjyx/Cross-LanguageNameBindingRecognition

[2]https://imbalanced-learn.org/stable/over_sampling.html

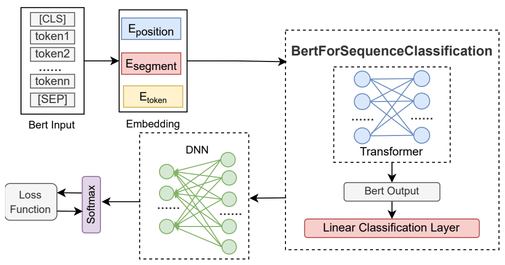

Fig. 2: The details of the name binding recognition model training component.

corresponding code, which can be expressed as follows. This token sequence will be input into the BERT model.

$$token = t_{file} + t_{identifier} + t_{code} \tag{1}$$

B. Name Binding Recognition Model Training

As shown in the Fig. 2, this component contains two steps, including token embedding and BERT fine-tuning model training.

a) Token Embedding: The token sequence obtained from the Data Preprocessing component is added two special tokens, i.e., the heading [CLS] in the front and the trailing [SEP] in the end respectively. The token embedding, the segment embedding that distinguishes two sentences, and the position embedding that encodes the position feature are the three embeddings to make up the input of the BERT model, which can be calculated as follows.

$$Input = E_{token} + E_{segment} + E_{position} \tag{2}$$

b) BERT fine-tuning model: Name binding recognition is regarded as a binary classification problem that can be broken down into the states of established binding and unestablished binding. To determine whether the non-Java identifier has a name binding relationship and to complete the name binding recognition, the BERT fine-tuning model is employed to summarize and analyze the deep code semantic of the name binding. The BERT fine-tuning model receives the processed vectors as inputs [9]. It combines a basic BERT Transformer [10] with an embedding layer followed by a series of identical self-attention blocks with a pooling layer, a dropout, and a linear classification layer. After that, our approach incorporates a DNN layer for additional categorization. Each vector that the BERT inputs constantly flows upward via a multi-layer decoder, passing through a feed-forward neural network and a self-attention mechanism at each layer. After encoding, a linear layer will be used for linear mapping, and a CLS vector fused with full-text semantic will be output at each place. The output layer will then output logits. The nonlinear activation function softmax function yields the probability output category, and CrossEntropyLoss function computes the loss.

C. Duplicate Name Binding Discrimination

After the name binding recognition model training, we design a further name binding discrimination component. The aim of this component is to discriminate the exact binded identifiers if there are duplicate names. It contains three steps, i.e., collecting context information, extracting classification features, and training duplicate name discrimination models.

A field may have duplicate ones with the same name in different classes. The previous component can decide the name binding for fields whose names only appear once. Hence, for those duplicate fields, it is impossible to bind all the ones with the same name, since a lot of improper binding of unrelated fields with the same name will be produced. We find that the differentiation between fields with duplicate names has a strong association with the programming context information in the same file as well as the framework rules employed in the project by observing name binding pairs. By examining the programming context information and the framework rules, this component extracts some features and then utilizes the ensemble learning to combine several classifiers to achieve the duplicate name binding discrimination.

a) Context information collection: As we described above, the programming context information is important for duplicate name binding discrimination. Hence, for the fields in Java, we first transform its corresponding class into an Abstract Syntax Tree (AST) [11]. Based on the formed AST, we then extract the programming context information of a specific field, including the class name, all the enclosed members (including other fields and all the methods) of this class, all the members in inherited class or interfaces, the comments, and the annotations. In our approach, we extract these programming context information using the popular JavaParser[3]. As for the identifiers in other languages, e.g., XML, we extract the all the tokens in the same hierarchy within the same tag of a specific identifier and regard these tokens as the context information. we employ the widely-used beautifulsoup[4] to extract the context information for those identifiers in other language.

b) Feature extraction: We design several features by examining the enclosed source code, the context information, and the framework help documentation, along with the observation of the frameworks and projects. These proposed features include the context similarity, the filename similarity, the occurrence, and the framework fundamental principles.

- Context similarity. For the context information of identifiers in both the Java language and the other language, we process them with the same natural language processing steps, including tokenization, stemming, and stop word removal. After that, the context information is transformed into a digital vector representation, in which each dimension represents a specific token and the corresponding value is calculated by the widely-used Term Frequency–Inverse Document Frequency (TF-IDF). To compare the context similarity of name binding pairs, we employ cosine similarity to calculate the similarity of the two context vectors. The more similar of the context

[3]https://github.com/javaparser/javaparser
[4]https://www.crummy.com/software/BeautifulSoup/bs4/doc/

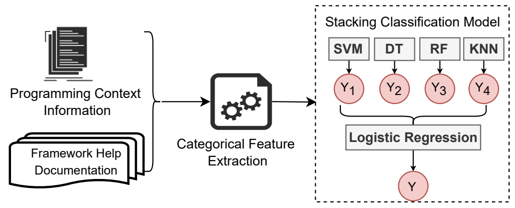
Stacking Classification Model
SVM
DT
RF
KNN
Y1
Y2
Y3
Y4
Logistic Regression
Y
Programming Context Information
Framework Help Documentation
Categorical Feature Extraction

Fig. 3: The workflow of the duplicate name binding discrimination component.

for a potential name binding pair, the more likely that the two identifiers are binded.

- Filename similarity. Given a potential name binding pair, we first remove the file type-specific suffix from the file name of the two identifiers. Since the file name of the two identifiers are usually short, to determine the degree of file name similarity, we choose the Levinstein ratio and the longest common subsequence length (LCS) as two separate features to measure the file name similarity of a potential name binding pair.
- Occurrence. This feature counts the occurrences and determines the probability that the identifier in other language presented in the Java source code file by comparing the class name of the field and other fields of the same name.
- Framework fundamental principles. This feature summarizes the fundamental guidelines for differentiating Java identifiers with the same name from the framework help documentation, such as the definition of properties related to Spring Bean in the spring framework.

c) Duplicate-name discrimination model training: In this component, the name binding pairs identified by the name binding recognition model are further judged. This component regards the duplicate name discrimination as a binary classification problem. The extracted features of each potential name binding pair are first normalized. Then, we employ an ensemble learning approach to achieve good performance by merging several classifiers together. As shown in the Fig. 3, several popular classifiers, including Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), and K-Nearest Neighbors (KNN), are chosen for the basic classifier [12]. The outputs (the probabilities belonging to each category) of these classifiers are further input into the Logistic Regression model, which will output the name binding discrimination for duplicate identifiers, i.e., whether the two identifiers need to be binded.

D. Cross-language Name Binding Prediction

In the prediction phase, given an identifier, we first discover all identifiers with the same name in the project using string matching. In such a way, we can form a series of potential name binding pairs by linking all the detected identifiers with this identifier. For each potential name binding pair, we extract the context of the two identifiers and feed them into the trained name binding recognition model. The trained model can be used to predict whether a specific name binding pair is correct or not. Furthermore, if the identifier has duplicate ones, we extract the programming context for identifiers with duplicate names and calculate the pre-defined features. By inputting these features into the trained duplicate name binding discrimination model, we can figure out the exact name binding. By detecting the name binding, it will be easy for developers to perform the identifier renaming and propagation.

III. Experimental Setup

A. Data Collection

In this study, we select 10 open source MLSAs that use Java as their primary programming language with at least one other language. These projects come from various domains, are programmed with different frameworks, and have various code scales. Hence, we think that the chosen projects are representative and can be used as the typical benchmark dataset for the cross-language name binding. Table I shows the characteristics of these selected projects.

The procedures of constructing the dataset includes collecting cross-language name binding pairs, collecting features, and manual annotation, which are illustrated as follows.

a) Collecting cross-language name binding pairs: For each project, we first extract the fields from all the Java classes and calculate how many times they appear in different classes. Then, we use a global search to find and locate all identifiers in the project with the same name. After that, we form a series of name binding pairs, which are composed of a field and an identifier with the same name.

b) Collecting features: For the fields in Java classes with duplicate names, it is important to specifically identify unrelated identifiers with the same name in order to generate correct name binding pair. Hence it is necessary to collect information on distinguishing attributes with duplicate names. We have explained the details of the extracted features for duplicate name binding discrimination. In this step, we collect the file name, the file path, the code line number, the context information for each name binding pair, which are used in the feature calculation.

c) Manual annotation: After collecting all of the potential name binding pairs and extracting the corresponding features, we recruit six volunteers to manually annotate the name binding dataset. A name binding pair is assigned to three volunteers to judge whether it is a correct name binding pair or not. These volunteers are all master students in software engineering, with at least four years in studying Java frameworks. Hence, we think that they can complete this annotation task. For a specific name binding pair, if at least two of the three volunteers think that it is correct, then this name binding pair is judged as a correct name binding pair. This annotation task takes one week to complete for each volunteer. Table I also shows annotation results. For example, we collect 501610 name binding pairs in total from the 10 projects, of which 12343 are correct name binding pairs. We can see that the correct name binding pairs only take up a tiny fraction (about 2.5%).

TABLE I: Details of the selected projects.

Project	Domain	Framework	Size	LOC	All fields	Correct binding	Incorrect binding
itracker	Issue Tracker	Spring, Hibernate, Struts	10940kB	110k	773	2014	94220
sagan	Reference Application	Spring	6547kB	20k	323	470	27892
springside	JavaEE Application	Spring	2038kB	28k	287	72	15542
Tudu-Lists	Todo Lsts Management	Spring, Hibernate	822kB	8k	76	92	2047
zksample2	ZK Framework Application	Spring, Hibernate	5262kB	35k	1067	402	15674
jrecruiter	Job Posting Solutions	Spring, Hibernate, Struts	7327kB	13k	163	112	8519
hispacta	Maven Web Application	Spring, Hibernate, Tapestry	183kB	3k	68	154	1719
powerstone	Java Workflow System	Spring, Hibernate, Struts	2448kB	31k	371	592	8760
jtrac	Issue Tracker	Spring, Hibernate, Wicket	2149kB	22k	328	284	20604
mall	E-commerce System	Spring, Springboot, MyBatis	9103kB	87k	600	8151	294290

B. Baselines

In the literature, there is no existing approach that aims to bind identifiers in multiple languages. Hence, in this study, we choose the following baselines for comparison and verify the performance of our approach.

a) The approach based on code similarity: Intuitively, the more similar of the enclosed code of the two identifiers, the more likely that the two identifiers are binded. Hence, this approach calculates the code similarity between the identifiers in the correct name binding and regards the median value as the threshold. For a test potential name binding, we calculate the code similarity between the two identifiers. If the code similarity is larger than the threshold, this name binding is judged as a correct name binding. Otherwise, it is judged as an incorrect name binding.

b) The approach based on context similarity: Similarly, this approach calculates the context similarity between the two identifiers in the name binding and regards the median value as the threshold. If the context similarity exceeds the threshold for a specific test name binding, it is judged as the correct one. Otherwise, it is judged as an incorrect one.

C. Evaluation Method and Metrics

We employ the widely-used ten-fold cross-validation as the evaluation method. Specifically, each project is divided into ten equal folds, nine of which are used for training in each experiment while the remaining one is used as the test set for evaluation. We repeat this process for ten times and calculate the average results as the final results. In terms of the evaluation metrics, we employ the frequently-used Precision, Recall, and F-Measure to evaluate the performance of different approaches.

D. Default Parameters

Our approach employs the pre-trained BERT with 12 hidden layers, a 768-dimensional tensor, 12 self-attentive heads, and 110M parameters in total. We choose the AdamW optimizer to fine-tune BERT, and set the Batch size to 16, the learning rate to 3e-5, the epochs to 6 by default.

IV. EXPERIMENTAL RESULTS

A. RQ1: What is the performance difference between our approach and the baseline approaches?

Motivation. To show the effectiveness of our approach, we compare it against the baseline approaches based on the constructed dataset. By exploring this RQ, we want to figure out whether our deep learning-based approach outperforms the existing approaches.

TABLE II: The results for unique identifier name binding recognition.

Unique Project	Precision(%)			Recall(%)			F-Measure(%)		
	Ours	Code	Context	ours	Code	Context	Ours	Code	Context
itracker	**89.12**	9.19	24.29	**93.33**	45.05	45.95	**91.08**	15.27	31.78
sagan	86.87	8.38	**94.74**	**97.86**	50.00	64.29	**91.99**	14.36	76.60
springside	**82.33**	0.78	42.86	**85.71**	57.14	42.86	**81.72**	1.53	42.86
Tudu-Lists	**64.58**	18.75	46.15	**81.82**	**81.82**	54.55	**70.98**	30.51	50.00
zksample2	**71.00**	50.00	70.00	**82.14**	50.00	50.00	**75.77**	50.00	58.33
jrecruiter	**88.93**	7.59	32.26	**84.44**	66.67	55.56	**84.25**	13.64	40.82
hispacta	81.62	13.39	**100.00**	**100.00**	56.67	54.55	**89.35**	21.66	70.59
powerstone	**81.71**	38.30	76.92	**98.06**	58.06	48.39	**88.61**	46.15	59.41
jtrac	**82.18**	12.79	60.00	**93.33**	81.48	66.67	**86.36**	22.11	63.16
mall	88.45	63.42	**99.49**	**94.49**	60.71	49.62	**91.33**	62.03	66.22
Average	**81.68**	22.26	64.67	**91.12**	60.76	53.24	**85.14**	27.73	55.97

TABLE III: The results of duplicate identifier name binding discrimination.

Repeat Project	Precision(%)			Recall(%)			F-Measure(%)		
	Ours	Code	Context	Ours	Code	Context	Ours	Code	Context
itracker	**98.86**	1.44	3.21	**90.48**	51.03	43.15	**94.47**	2.80	5.97
sagan	**96.85**	1.55	11.68	**96.97**	52.24	34.33	**96.86**	3.02	17.42
springside	**73.06**	0.45	8.62	**70.00**	50.00	62.50	**71.30**	0.89	15.15
Tudu-Lists	**73.33**	4.63	10.26	**60.00**	62.50	50.00	**65.55**	8.62	17.02
zksample2	**100.00**	10.40	27.18	**96.98**	73.58	52.83	**98.47**	18.22	35.90
jrecruiter	**92.67**	0.27	2.19	96.00	40.00	**100.00**	**94.18**	0.54	4.29
hispacta	60.00	**66.67**	**66.67**	60.00	**100.00**	**100.00**	60.00	**80.00**	**80.00**
powerstone	**95.96**	23.53	30.19	**97.89**	56.14	56.14	**96.90**	33.16	39.26
jtrac	**93.16**	1.12	5.15	**86.45**	58.06	48.39	**89.31**	2.20	9.32
mall	**98.97**	2.83	48.20	**98.43**	56.65	52.03	**98.69**	5.38	50.04
Average	**88.29**	11.29	21.33	**85.32**	60.02	59.94	**86.57**	15.48	27.44

Approach. For the two baseline approaches, we implement them by ourselves. To guarantee the implementation, we employ the code review mechanism. Notably, we use the same dataset divisions for training and prediction for both our approach and the baseline approaches. In such a way, we can make a fair comparison. We employ the ten-fold cross-validation to compare our approach and the baseline approaches.

Results. To thoroughly evaluate the performance of different approaches, we compare our approach against the baseline approaches from two aspects, i.e., unique identifier name binding recognition and duplicate identifier name binding discrimination. For the unique identifiers, which only appear once in the projects, we need to perform the name binding recognition. Furthermore, for those identifiers with duplicate names, we need to additionally perform the name binding discrimination. Table II and Table III show the two detailed results respectively, where *code* and *context* means the two baseline approaches using code and context similarity.

For the unique identifiers, we can see from Table II that

TABLE IV: The comparison results of our approach and its variants.

Project	Precision(%)			Recall(%)			F-Measure(%)		
	Ours	Variant1	Variant2	Ours	Variant1	Variant2	Ours	Variant1	Variant2
itracker	**97.21**	85.28	88.54	**97.14**	18.90	87.94	**97.15**	30.72	88.17
sagan	95.87	52.89	**97.35**	96.95	79.08	**97.61**	96.32	62.88	**97.43**
springside	**100.00**	89.17	96.67	**98.00**	44.17	95.50	**98.89**	57.33	95.46
Tudu-Lists	**98.00**	63.33	95.50	85.83	29.17	**90.50**	89.13	38.40	**92.42**
zksample2	99.64	**100.00**	**100.00**	**99.37**	96.05	95.40	**99.50**	97.91	97.56
jrecruiter	**86.67**	10.00	70.83	75.00	5.00	**82.50**	**76.33**	6.67	71.17
hispacta	**60.00**	**60.00**	50.00	**60.00**	**60.00**	50.00	**60.00**	**60.00**	50.00
powerstone	98.65	**98.93**	93.09	**98.24**	69.92	92.25	**98.42**	81.27	92.47
jtrac	92.88	**100.00**	94.67	91.88	43.39	**93.43**	92.12	59.81	**93.87**
mall	**100.00**	84.44	99.80	**99.98**	97.44	99.45	**99.99**	90.47	99.63
Average	**92.89**	74.40	88.65	**90.24**	54.31	88.46	**90.79**	58.54	87.82

our approach achieves the best Precision than the two baseline approaches in 7 out of 10 projects. In terms of the Average Precision in the 10 projects, our approach achieves 81.68%. In contrast, the two baseline approaches only achieve 22.26% and 64.67% respectively. In terms of Recall and F-Measure, we can find that our approach achieves the best result in all the 10 projects. Our approach achieves the average Recall of 91.12%. However, *code* and *context* only achieve the average Recall of 60.76% and 53.24%. As for the average F-Measure, our approach achieves 85.14%, which outperforms *code* and *context* by 57.41% and 29.17% respectively. Hence, we can see that our approach significantly improves the results of unique identifier name binding recognition compared against the two baseline approaches.

For those identifiers with duplicate names, our approach can also achieve satisfactory results as shown in Table III. In terms of the Precision, we can see that our approach achieves the best Precision in 9 out of 10 projects, with an average Precision of 88.29%. However, *code* and *context* only achieve the average Precision of 11.29% and 21.33% respectively. It shows that the two baseline approaches cannot accurately discriminate the name binding for duplicate identifiers. In terms of Recall, we can see that our approach achieves the best results not only in each single project, but also in the average result (i.e., 85.32%). In contrast, *code* and *context* achieve the average Recall of 60.02% and 59.94%. As for F-Measure, our approach achieves the average result of 86.57% in the 10 projects. However, the two baseline approaches only achieve 15.48% and 27.44%. It means that the two baseline approach cannot discriminate name binding for duplicate identifiers.

When comparing the two baseline approaches, we can see that *context* achieves better results than *code* in both unique identifier name binding recognition and duplicate identifier name binding discrimination. For example, in terms of the synthetic metric F-Measure, *context* performs better than *code* in unique identifier name binding recognition (55.97% compared to 27.73%) and duplicate identifier name binding discrimination (27.44% compared to 15.48%). It shows that the context information is useful in name binding for identifiers.

We can also find our approach does not perform better results than the baselines in the Hispacta project in terms of the F-Measure. This is because this project contains the least number of correct and incorrect binding as shown in Table I. In addition, the Hispacta project only includes the name binding for 6 duplicate identifiers, which are both insufficient and inconsistent. Deep learning models need a lot of training data. If the model is not fully trained, the performance may suffer.

Conclusion. Our proposed name binding recognition and recognition approach significantly performs better than the baseline approaches.

B. RQ2: How effective is the context information in improving the performance of our approach for name binding?

Motivation. To accurately discriminate name binding for duplicate identifiers, we extract the context information of identifiers as features. To show the effectiveness of employing the context information in our approach, we set up this RQ.

Approach. We define two variants to compare against our original approach. The first variant only removes the context information based features and remain the other features the same to discriminate name binding for duplicate identifiers. In the second variant, we change the definition of the context information. Specifically, the second variant employs all words as the context nformation for extracting the corresponding features, excluding keywords and tagged values. Hence, the first variant removes the context information and the second variant employs a redundant context information. By comparing the two variants against our original approach, we can figure out whether the context information used in our approach is effective.

Results. The comparison results between our approach and its variants are shown in the Table IV. First, we can see that the performance of our approach is improved by combining the two types of identifiers. For example, our approach achieves the average Precision of 92.89%, the average Recall of 90.24%, and the average F-Measure of 90.79%. All of these evaluation metrics exceeds 90%. When we compare our approach against the two variants, we find that our approach achieves the best results. For example, our approach achieves the average Precision of 92.89%. However, the first variant only achieves 74.40% and the second variant achieves 88.65%. In terms of Recall, our approach achieves the average result of 90.24%, which outperforms the two variants by 35.93% and 1.78%. It means that changing the context information from our approach will have an negative impact on the Recall value. As for the average F-Measure, we can see that our approach achieves 90.79%. However, the two variants achieve

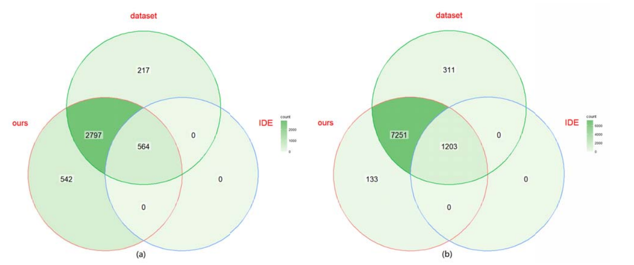

Fig. 4: The comparison results among the golden dataset, our approach, and IntelliJ IDEA. (a) is for unique identifiers. (b) is for duplicate identifiers.

58.54% and 87.82% respectively. From the results, we can find that either removing the context information from our approach or changing another type of the context information, the performance of our approach will decline. It means that the context information is effective in our approach.

Conclusion. The context information used by our approach is helpful for its performance.

C. RQ3: Can our approach really help developers bind identifiers in the real scenario?

Motivation. In this RQ, we figure out whether our approach can really help developers in binding cross-language identifiers compared to the widely-used Integrated Development Environment (IDE). If it is true, our approach can be helpful for developers in the real software development process.

Approach. We employ the widely-used IntelliJ IDEA for comparison in this RQ. Specifically, for the manually annotated golden dataset, we count and examine how many name bindings can our approach and the IntelliJ IDEA detect for both unique identifiers and duplicate identifiers. The IntelliJ IDEA provides the *Edit-Refactor-Rename* function, and we use this function to rename the detected identifiers by IntelliJ IDEA. If there is an error, it means that the name binding for this specific identifier is incorrect. Otherwise, the name binding for the specific identifier is correct. In such a way, we can calculate and compare the exact correct number of name binding (the same as the golden dataset) for unique and duplicate identifiers in terms of our approach and IntelliJ IDEA.

Results. The Fig. 4 shows the comparison results among the golden dataset, our approach and IntelliJ IDEA, wheter (a) shows the results of unique identifiers and (b) shows the results of duplicate identifiers. We can see that our approach could detect 2797+564=3361 name binding pairs for unique identifiers. In contrast, IntelliJ IDEA can only detect 564 name binding pairs for unique identifiers. Compared against IntelliJ IDEA, our approach can significantly detect more correct name binding pairs. The correctly detected name binding pairs of our approach is almost 6 times as large as that of IntelliJ IDEA. In terms of duplicate identifiers, we can also find the similar phenomenon. For example, our approach can detect 7251+1203=8454 name binding pairs for duplicate identifiers. In contrast, IntelliJ IDEA can only detect 1203, which is far less than that of our approach. From this results, we can find that our approach can detect name binding pairs more accurately than IntelliJ IDEA.

Conclusion. Our approach can accurately detect name binding compared to IntelliJ IDEA, showing that our approach is helpful for developers in the real scenario.

V. Threats to Validity

Threats to internal validity. In this study, we configure the deep learning model as shown in subsection III-D. These parameters may have an influence on the performance of our approach. Actually, we have conducted several preliminary experiments to adjust these parameters to find the good or suitable values for them. Hence, we think that this threat has been taken into consideration and reduced as much as possible.

Threats to external validity. To validate the effectiveness of our approach, we select 10 popular projects with multiple-language to construct the dataset. However, because our approach is based on the deep learning model, which greatly depends on the scope of the dataset, we cannot guarantee that our findings can be appliable to other projects. For the projects with a small scope or few name binding pairs, the effectiveness of our approach is still unknown. In the future, we will employ more datasets to validate our approach.

VI. Related Work

In the literature, the majority of the existing approaches focus on the name binding inside in a single language. In order to identify copy-and-paste activities performed by developers in Java programs and to automatically rename identifiers, Jablonski and Hou [13, 14] propose the *CReN* approach for the single language name binding based on the clustering of AST relationships. By developing globally unique IDs, De Jonge et al. [15], Guo et al. [16], and Schäfer et al. [17] provide name binding assistance. All identifiers in the source code are given globally unique reference names by De Jonge et al. [15], who also assign the same reference name to two identifiers that are connected to the same declaration. Guo et al. [16] give entities in Java applications distinct identifiers and use tags to connect those identifiers to entity references. With the help of AST, Schäfer et al. [17] generate symbolic names and bind them using an inverted name lookup table. By analyzing static data to group object properties for the dynamic language JavaScript, Feldthaus et al. [8, 18] approximate the name binding in dynamic language renaming. The aforementioned approach are all language-specific or within a single language. However, given that the majority of modern software are multiple-language software applications, simply binding identifiers within a language is not enough.

Existing studies on cross-language name binding in MLSAs all use reference-based techniques, which discover all the references to the identifiers consistently throughout the project [4]. These techniques are limited to only two languages (one is the programming language like Java and the other one is the domain-specific language like XML), Hence, they are still not appliable to the modern software. In addition, the reference-based techniques also cannot be generalized to other languages. Hence, it is very necessary and important to propose an universal cross-language name binding approach.

VII. CONCLUSION AND FUTURE WORK

Binding identifiers in multiple languages is crucial for modern software development, especially when developers perform the identifier renaming and propagation. Existing studies only focus on the name binding for a specific language without considering the cross-language name binding. Hence, in this study, we propose a cross-language name binding approach for the Java frameworks based on the deep learning model. This novel approach determines the name binding by string matching, combining context information, and framework fundamental rules. We evaluate our approach on 10 multiple-language projects. Experimental results show that our approach can achieve the average F-Measure of 85.14% for unique identifiers and 86.57% for duplicate identifiers.

In the future, we will further validate our approach on more projects with various frameworks and sizes. We will also extend our approach to other granularities of identifiers. In addition, we will develop an automatic tool or plugin encapsulating our approach and release it into the public to help developers in real-time.

ACKNOWLEDGMENT

This work is partially supported by the National Natural Science Foundation of China under Grant No. 62272225, the Fund of Prospective Layout of Scientific Research for NUAA (Nanjing University of Aeronautics and Astronautics), and the Postgraduate Research Practice Innovation Program of NUAA under Grant No. xcxjh20221613.

REFERENCES

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

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

[3] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J Decker, and Christian D Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 26–33, 2018.

[4] Guangjie Li, Hui Liu, and Ally S Nyamawe. A survey on renamings of software entities. *ACM Computing Surveys (CSUR)*, 53(2):1–38, 2020.

[5] Philip Mayer and Andreas Schroeder. Towards automated cross-language refactorings between java and dsls used by java frameworks. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pages 5–8, 2013.

[6] Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Yann-Gaël Guéhéneuc, Naouel Moha, Geoffrey Hecht, Ghizlane El Boussaidi, and Jean Privat. Static code analysis of multilanguage software systems. *arXiv preprint arXiv:1906.00815*, 2019.

[7] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology*, 140:106675, 2021.

[8] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. *ACM SIGPLAN Notices*, 48(10):323–338, 2013.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[11] Roya Hosseini and Peter Brusilovsky. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, volume 1009, pages 60–63. University of Pittsburgh, 2013.

[12] Shuo Feng, Jacky Keung, Xiao Yu, Yan Xiao, and Miao Zhang. Investigation on the stability of smote-based oversampling techniques in software defect prediction. *Information and Software Technology*, 139:106662, 2021.

[13] Patricia Jablonski and Daqing Hou. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, 2007.

[14] Patricia Jablonski and Daqing Hou. Renaming parts of identifiers consistently within code clones. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 38–39. IEEE, 2010.

[15] Maartje De Jonge and Eelco Visser. A language generic solution for name binding preservation in refactorings. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, pages 1–8, 2012.

[16] Xinping Guo, James R Cordy, and Thomas R Dean. Unique renaming of java using source transformation. In *Proceedings IEEE International Workshop on Source Code Analysis and Manipulation*, pages 151–160, 2003.

[17] Max Schäfer, Torbjörn Ekman, and Oege De Moor. Sound and extensible renaming for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 277–294, 2008.

[18] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 119–138, 2011.