Automated Software Entity Matching Between Successive Versions

Bo Liu  
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China  
liubo@bit.edu.cn

Hui Liu*  
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China  
liuhui08@bit.edu.cn

Nan Niu  
Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA  
nan.niu@uc.edu

Yuxia Zhang  
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China  
yuxiazh@bit.edu.cn

Guangjie Li  
National Innovation Institute of Defense Technology, Beijing, China  
liguangjie.er@126.com

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

Abstract—Version control systems are widely used to manage the evolution of software applications. However, such version control systems take source code as lines of plain text, and thus they cannot present the evolution of software entities embedded in the source code. To this end, a few approaches have been proposed to match software entities before and after a given commit, known as software entity matching algorithms [2]–[4]. Notably, matching software entities not only facilitates developers (maintainers) to comprehend the evolution of software applications, but it also serves as the basis for other automated software engineering tasks, e.g., refactoring discovery [5]–[7] and entity tracking [8]–[10]. Consequently, any mistakes (false positives and false negatives) made by software entity matching algorithms may significantly affect the performance of the subsequent automated software engineering tasks that rely heavily on them.

Entity matching originated from “origin analysis” that was opened up by Godfrey et al. [11]–[13]. Existing entity matching algorithms could be divided into two categories. The first category focuses on the evolution of system designs, and matches design entities without considering their low-level implementation (e.g., method bodies). For example, the well-known UMLDiff proposed by Xing and Stroulia [3]. It recovers UML class diagrams from the source code, and maps nodes (entities) between two class diagrams that represent the source code before and after a revision (commit). Such high-level design-based matching is lightweight and thus highly efficient. The second category fully considers the implementation of software entities. For example, RefactoringMiner proposed by Tsantalis et al. [7], [14]. It represents the implementation of software entities as abstract syntax trees (ASTs), and computes the similarity between two entities according to the name-based similarity and the statement-based similarity. With such similarities, it maps entities between two successive versions, and leverages a sequence of heuristics to discover software refactorings (e.g., rename class and extract method refactorings) based on the mapping. Although evaluation results suggest that existing entity matching algorithms are often accurate [14], the number of mismatched entities remains non-negligible (see Section III for details).

To this end, in this paper, we propose a novel matching algorithm, called ReMapper, to improve the accuracy in

* Corresponding author

I. INTRODUCTION

Software entity matching is the cornerstone of computer-aided comprehension of software evolution [1]. Although version control systems are widely used to manage the evolution of software applications, such version control systems usually take source code as lines of plain text, and thus they cannot comprehend the evolution of software entities embedded in the source code. To this end, a few approaches have been proposed to match software entities before and after a given commit,
mapping entities between two successive versions. The key insight of ReMapper is that the qualified name, the implementation, and the references of a software entity together can distinguish it from others. Consequently, it computes the name-based similarity, the implementation (AST) based similarity, and the reference-based similarity, respectively. Note that the references of an entity (i.e., entities accessing it) represent the external attributes of the entity, and thus they often survive various inner changes to the entity, e.g., changes to its implementation and changes to the entity name. To the best of our knowledge, ReMapper is the first entity matching algorithm that takes full advantage of all such three categories of similarities. Although design-based matching algorithms like UMLDiff leverage the references of entities, the references are often simply taken as connections between entities, (i.e., edges in class diagrams), and they are not distinguished from other connections (e.g., composition). Besides, design-based matching algorithms do not exploit the implementation-based similarity as ReMapper does. Although more recent matching algorithms like RefactoringMiner exploit the implementation-based similarity, they do not take advantage of the reference-based similarity. Notably, ReMapper is iterative because the mapping depends on the reference-based similarity whereas the reference-based similarity depends on the mapping of entities as well. The iterative matching differs from existing ones (e.g., the iterative matching in UMLDiff) in that it not only matches unmatched entities, but also re-matches entities that have already been temporarily matched in the previous iterations. The iteration terminates until the mapping becomes stable, i.e., further iterations would not change the mapping anymore.

We evaluated ReMapper on a benchmark consisting of 215 commits from 21 real-world projects. The evaluation results suggest that ReMapper substantially outperformed the state of the art in entity matching, reducing the number of mistakes (false positives plus false negatives) substantially by 85.8%. Since entity matching algorithms serve as the cornerstone of automated discovery of software refactorings [6], [7], we also evaluated to what extent ReMapper may improve the automated discovery of software refactorings. We replaced the entity matching algorithm in RefactoringMiner (that represents the state of the art in automated discovery of refactorings) but kept the other parts (i.e., the heuristics-based rules for detecting refactorings) intact. Our evaluation results suggest that ReMapper substantially improved the state of the art in automated discovery of software refactorings. It improved the recall in refactoring discovery by 6.9% and reduced the number of false positives by 72.6%.

In this paper, we make the following contributions:

• We propose a novel entity matching algorithm (called ReMapper) that takes full advantage of name-based similarity, implementation-based similarity, and reference-based similarity.

• We construct a new dataset for entity matching (called em-Dataset), validated by multiple developers [15]. Our evaluation on the dataset validates that ReMapper can substantially reduce the number of false positives and false negatives.

• We boost the automated discovery of refactorings by the proposed entity matching algorithm, resulting in a new refactoring miner (called RefactoringExtractor).

II. APPROACH

A. Overview

Fig. 1 presents an overview of ReMapper. It takes as input a Java project repository and a commit ID, and returns a mapping between software entities before and after the commit. Given the repository and commit ID, ReMapper works as follows:

• First, it retrieves a list of files involved in the commit, i.e., files that have been removed, modified, or newly added by the commit. From the retrieved files, it retrieves all to-be-matched Java entities, and identifies intact entities (noted as IntactE) that have not been changed by the given commit. It also leverages a qualified name-based matching algorithm to match other entities, and the resulting mapping is noted as $M$.

• Second, it leverages an implementation-based matching algorithm to initially match entities that have not yet been matched so far, and the resulting initial mapping is noted as $M'$.

• Third, it leverages a reference-based iterative matching algorithm to match (or re-match) entities (excepted for intact entities in IntactE and matched entities in $M$). The iteration is composed of two steps. In the first step, it computes (or re-computes) the reference-based similarity between entities based on the initial mapping $M'$. In the second step, it matches (or re-matches) entities according to the reference-based similarity, and updates mapping $M'$. The iteration terminates when $M'$ becomes stable.

![Fig. 1. Overview of ReMapper](image-url)
newly added files are added to set \(\text{ignore} \) intact files and non-source code files \([7], [14]\). All changes (denoted as \(n\)) is added to the new version \((V_{n+1})\) of the same document that was modified by the commit. Notably, entity matching algorithms usually ignore intact files and non-source code files \([7], [14]\). All newly added files are added to set \(AD\), and all removed files are added to set \(RD\). The modified files are recorded by a tuple \(<n, n'>\) where \(d\) and \(d'\) represent the old version \((V_n)\) and the new version \((V_{n+1})\) of the same document that was modified by the given commit.

From each tuple \(<n, n'>\in MD\), we retrieve all Java entities declared in \(d\) and \(d'\), respectively. All entities declared in \(d\) are added to a set (noted as \(oldE\)), and entities declared in \(d'\) are added to another set (noted as \(newE\)). Notably, the proposed approach only matches methods, fields, classes, interfaces, enums, @interface (annotation type declarations), initializers, enum constants, and annotation members. It does not match Java entities within methods (e.g., parameters, variables, and statements), and thus such entities are ignored. Notably, the number of entity types handled by the proposed approach is significantly greater than that handled by existing approaches. For example, the well-known software entity matching algorithm \([4]\) matches method-level software entities only.

We identify intact entities in \(d\) by matching entities in \(oldE\) against entities in \(newE\). Entity \(oe\in oldE\) matches entity \(ne\in newE\) if and only if:

- \(oe \text{ and } ne\) are of the same entity type;
- \(oe \text{ and } ne\) share identical fully qualified name; and
- \(oe \text{ and } ne\) share identical implementation.

The implementation of an entity depends on its entity type. For example, the implementation of a method includes the signature and its method body whereas the implementation of a field includes its complete declaration (including its initialization). We retrieve the implementation of an entity by using ASTNode.toString() \([17]\) on it. The comparison of entities’ implementations is a pure text-based comparison and thus any changes in the implementation will prevent the matching of entities in this step. If \(oe\) and \(ne\) match, we add a tuple \(<oe, ne>\) to set \(IntactE\), and remove \(oe\) from \(oldE\) and \(ne\) from \(newE\). In Fig. 2, field \(type\) and method \(handleConstructor\) are added to \(IntactE\) because their fully qualified names and implementations keep intact (without any changes) in the given commit.

For the remaining entities in \(oldE\) and \(newE\) that we fail to match in the preceding step, we try to match them again, ignoring their implementation. That is, entity \(oe\in oldE\) matches entity \(ne\in newE\) if and only if:

- \(oe\) and \(ne\) are of the same entity type;
- \(oe\) and \(ne\) share identical fully qualified name;
- \(oe\) and \(ne\) share the same signature if they are methods; and
- \(oe\) and \(ne\) share the same data type if they are fields or annotation members.

This matching step may help identify entities whose implementation has been modified by the given commit but their identity (including fully qualified names, method signatures, and data types) is kept intact. If \(oe\) and \(ne\) match, we add a tuple \(<oe, ne>\) to set \(M\), and remove \(oe\) from \(oldE\) and \(ne\) from \(newE\). In Fig. 2, class \(Processor\) and method \(verifyParameterNamesAreAvailable\) are added to \(M\) because their identities (without considering their internal implementation) keep intact in the given commit.

**C. Implementation-based Initial Matching**

Entities to be matched are classified into two categories. The first category is composed of Java entities that have the potential to contain other to-be-matched entities. For example, classes have the potential to contain methods and fields that should be matched as well. The other category is composed of Java entities that cannot contain other to-be-matched entities.
Algorithm 1: Initial Matching of Atomic Entities

```
Input : $2bmE_n$ and $2bmE_{n+1}$
Output: $\mathcal{M}'$
1 $\mathcal{L} \leftarrow \emptyset$
2 foreach $oe \in 2bmE_n.atomicEntities$ do
3     foreach $ne \in 2bmE_{n+1}.atomicEntities$ do
4         if $oe.type == ne.type$ then
5             $\text{implSim}(oe, ne) = \text{compImplSim}(oe, ne)$
6             if $\text{implSim}(oe, ne) \geq 0.5$ then
7                 $\mathcal{L}.add(\langle oe, ne, \text{implSim}(oe, ne) \rangle)$
8             end
9         end
10     end
11 end
12 sortBySim($\mathcal{L}$) // in descending order
13 Function $\text{similarityBasedMatching}(\mathcal{L})$
14     $\mathcal{M}' \leftarrow \emptyset$, $\mathcal{O} \leftarrow \emptyset$, $\mathcal{N} \leftarrow \emptyset$
15     for $i = 0$; $i < \mathcal{L}.length$; $i++$ do
16         if $\mathcal{L}[i].oe \notin \mathcal{O}$ and $\mathcal{L}[i].ne \notin \mathcal{N}$ then
17             $\mathcal{M}'.add(\langle \mathcal{L}[i].oe, \mathcal{L}[i].ne \rangle)$
18             $\mathcal{O}.add(\mathcal{L}[i].oe)$
19             $\mathcal{N}.add(\mathcal{L}[i].ne)$
20         end
21     end
22 return $\mathcal{M}'$
23 end
```

For example, methods belong to this category because the proposed approach does not match Java entities within a method (like parameters, variables, and statements). For the sake of simplicity, we call the two categories compound entities and atomic entities, respectively.

All to-be-matched entities (except for intact entities in $\text{IntactE}$ and matched entities in $\mathcal{M}$) in $V_n$ and $V_{n+1}$ are noted as $2bmE_n$ and $2bmE_{n+1}$, respectively. We initially match atomic entities by Algorithm 1. For each atomic entity $oe \in 2bmE_n$, we compare it against each atomic entity $ne \in 2bmE_{n+1}$. If they are of different entity types, they are deemed unmatched. For example, we cannot map a field in the old version to a method in the new version. If they are of the same entity type, on Line 5 in Algorithm 1, we retrieve their corresponding ASTs and compute the implementation-based similarity (i.e., $\text{implSim}(oe, ne)$) between them. We add a triple $\langle oe, ne, \text{implSim}(oe, ne) \rangle$ to list $\mathcal{L}$ (Line 7) if and only if:

- $oe$ and $ne$ are of the same entity type; and
- $\text{implSim}(oe, ne) \geq 0.5$ (i.e., more than half of their AST nodes are common).

The triple suggests that $oe$ and $ne$ have the potential to be matched. We compute the implementation-based similarity of two atomic entities with dice coefficient [18] that is widely used to measure the similarity between two ASTs [19]–[21].

The implementation-based similarity is computed as follows:

$$\text{implSim}(e_1, e_2) = \text{dice}(t_1, t_2) = 2 \times \frac{|\text{nodes}(t_1) \cap \text{nodes}(t_2)|}{|\text{nodes}(t_1)| + |\text{nodes}(t_2)|},$$

(1)

where $t_1$ and $t_2$ are the ASTs associated with entities $e_1$ and $e_2$, respectively. $\text{nodes}(t_1)$ and $\text{nodes}(t_2)$ denote the AST nodes in $t_1$ and $t_2$, respectively. $\text{nodes}(t_1) \cap \text{nodes}(t_2)$ denotes the common nodes between the two sets. Nodes $nd_i \in \text{nodes}(t_1)$ and $nd_j \in \text{nodes}(t_2)$ represent a common node if they share the same node type (i.e., $nd_i.getNodeType() == nd_j.getNodeType()$) and equivalent value (i.e., $nd_i.toString() == nd_j.toString()$).

We sort $\mathcal{L}$ by implementation-based similarity in descending order (Line 12). Function $\text{similarityBasedMatching}$ (Lines 14–24) matches atomic entities to maximize the similarity between matched entities whereas each entity matches no more than one entity.

The initial matching of compound entities is exactly the same as the matching of atomic entities except that the similarity of compound entities is computed as follows:

$$\text{implSim}(e_1, e_2) = 2 \times \frac{|\text{subE}(e_1) \cap \text{subE}(e_2)|}{|\text{subE}(e_1)| + |\text{subE}(e_2)|},$$

(2)

where $\text{subE}(e)$ denotes all entities contained in $e$. For example, if $e$ is a class, $\text{subE}(e)$ is composed of all methods, fields, and inner types (e.g., inner classes and inner interfaces) declared within $e$. $\text{subE}(e_1) \cap \text{subE}(e_2)$ denotes the common entities within $\text{subE}(e_1)$ and $\text{subE}(e_2)$. The two entities $e_1 \in \text{subE}(e_1)$ and $e_2 \in \text{subE}(e_2)$ represent a single common entity if and only if $e_1$ and $e_2$ have been matched (or temporarily matched), i.e., $\langle e_1, e_2 \rangle \in (\text{IntactE} \cup \mathcal{M} \cup \mathcal{M}')$.

Because the similarity of two compound entities as defined in Equation 2 depends on the matching of their sub-entities, the initial matching should take a bottom-up strategy.

In the example depicted in Fig. 2, two atomic entities (field $\text{PARAMETER_NAME_DISCOVERER}$ on Line 112 in $V_n$ and field $\text{parameterNameDiscoverer}$ on Line 115 in $V_{n+1}$) are initially matched because they share more than half of their AST nodes, and they are added to $\mathcal{M}'$.

D. Reference-based Iterative Matching

As suggested by the overview in Fig. 1, reference-based iterative matching is a loop containing two iterative steps, i.e., computation of reference-based similarity and reference-based matching. We explain the loop (iteration) first, and then the two key steps within the loop.

1) Iterative Matching: The proposed approach leverages Algorithm 2 to conduct the reference-based iterative matching. In each iteration, the approach first (Lines 2-12) computes/updates the reference-based similarity based on the temporary mapping (i.e., $\mathcal{M}'$) generated by the previous iteration. With the updated similarity, the approach (Lines 13-19) rematches temporarily matched entities (as well as unmatched ones), and updates $\mathcal{M}'$ if it is necessary. The algorithm terminates when the mapping $\mathcal{M}'$ survives the latest iteration, i.e., the iteration does not bring any changes to $\mathcal{M}'$.
Algorithm 2: Reference-based Iterative Matching

Input : $2bmE_n$, $2bmE_{n+1}$, $\text{IntactE}$, $M$, and $M'$
Output: $M'$

1. while true do
   2. $\mathcal{L} \leftarrow \emptyset$
   3. foreach $e_1 \in 2bmE_n$ do
      4. foreach $e_2 \in 2bmE_{n+1}$ do
         5. if type-compatible($e_1$, $e_2$) then
            6. $\text{sim}(e_1, e_2) = \text{compSim}(e_1, e_2, \text{IntactE}, M, M')$
            7. if $\text{sim}(e_1, e_2) \geq 0.5$ then
               8. $\mathcal{L}.\text{add}(<e_1, e_2, \text{sim}(e_1, e_2)>)$
            9. end
        10. end
    11. end
   12. sortBySime(\mathcal{L}) // in descending order
   13. $S \leftarrow \text{similarityBasedMatching}(\mathcal{L})$
   14. if $M' == S$ then // no change on mapping
      15. return $M'$ // terminate loop
   16. else
      17. $M' \leftarrow S$
   18. end
 19. end

2) Computation of Reference-based Similarity: For each pair of entities $e_1 \in 2bmE_n$ and $e_2 \in 2bmE_{n+1}$, we compute their reference similarity as follows:

\[
\text{refSim}(e_1, e_2) = \frac{2 \times |\text{callers}(e_1) \cap \text{callers}(e_2)|}{|\text{callers}(e_1)| + |\text{callers}(e_2)|},
\] (3)

where $\text{callers}(e)$ denotes Java entities that access $e$. $\text{callers}(e_1) \cap \text{callers}(e_2)$ denotes entities accessing both $e_1$ and $e_2$. Notably, $e_1$ and $e_2$ represent an entity accessing both $e_1$ and $e_2$ if and only if $e_1 \in \text{callers}(e_1), e_2 \in \text{callers}(e_2)$, and $e_1$ has been matched (or initially matched) to $e_2$ (i.e., $<e_1, e_2> \in \text{IntactE} \cup M \cup M'$).

The reference-based similarity between $e_1$ and $e_2$ is the average of their reference similarity and implementation-based similarity:

\[
\text{sim}(e_1, e_2) = \frac{\text{refSim}(e_1, e_2) + \text{implSim}(e_1, e_2)}{2}
\] (4)

3) Reference-based Matching: To match entities according to the reference-based similarity defined in Equation 4, Algorithm 2 (Lines 3-12) computes the reference-based similarity between each type-compatible entity pair $<e_1, e_2>$ (Line 5). If two entities are of the same entity type, they are surely type compatible. Following Tsantalis et al. [7], [14], we also take classes, interfaces, and enums as type-compatible to each other because developers often switch among such types, e.g., changing an interface into a class. If we do not try to match interfaces against classes, we can never identify such refactorings. The same is true for enums.

If the reference-based similarity between $e_1$ and $e_2$ is greater than or equal to a preset threshold (empirically set to 0.5), we add the item $<e_1, e_2, \text{sim}(e_1, e_2)>$ to list $\mathcal{L}$ (Line 8), suggesting that they have the potential to match each other. We sort items in $\mathcal{L}$ by their reference-based similarity in descending order (Line 13). In case two items have equivalent reference-based similarity, we sort them according to the name-based similarity in descending order. The name-based similarity between $e_1$ and $e_2$ is the $n$-gram similarity [22] of their fully qualified names:

\[
\text{nameSim}(e_1, e_2) = \frac{2 \times |2g(\text{n-gram}(e_1)) \cap 2g(\text{n-gram}(e_2))|}{|2g(\text{n-gram}(e_1))| + |2g(\text{n-gram}(e_2))|},
\] (5)

where $\text{n-gram}(e)$ is the fully qualified name of $e$, and $2g(\text{n-gram}(e))$ denotes the set of 2-grams within $\text{n-gram}(e)$.

After sorting $\mathcal{L}$, the proposed approach rematches entities by using function $\text{similarityBasedMatching}$ on Line 14. Notably, this function has been defined in Algorithm 1.

In the example depicted in Fig. 2, after one iteration of matching, field $\text{PARAMETER_NAME_DISCOVERER}$ on Line 112 in $V_n$ and field $\text{parameterNameDiscoverer}$ on Line 115 in $V_{n+1}$ are matched and added to the temporary mapping ($S$) because they are accessed by the same method $\text{verifyParameterNamesAreAvailable}$. Then, the algorithm terminates and returns all matching pairs because $S$ equals the previous mapping ($M'$).

III. EVALUATION

The evaluation investigates the following research questions:

RQ1. Can ReMapper improve the state of the art in matching software entities between successive versions?

RQ2. How does the patch size influence the performance of ReMapper?

RQ3. Is ReMapper scalable?

To answer RQ1, we compared ReMapper against RefactoringMiner [14]. Although it was originally designed for refactoring discovery, its novel entity matching algorithm serves as its cornerstone (and its major contribution). We employed its entity matching algorithm only as the baseline. Pure entity matching algorithms were not employed for comparison because they were rather old (published more than fifteen years ago), and more recent advances in this line are often closely coupled to specific tasks, like discovery of refactorings [14], [23] and extraction of fine-grained code changes [21]. RefactoringMiner [14] is the latest advance in this line. For the sake of integrity, however, we also compared ReMapper against the pure entity matching algorithm JDiff [24], and published the results online [25].

RQ2 concerns whether and to what extent the performance of ReMapper may be influenced by the patch size (i.e., in a commit, the number of deleted or modified lines plus the number of added lines). It is likely that the larger the change (patch) is, the more challenging the entity matching is. By answering RQ2, we intend to examine the hypothesis. RQ3 concerns the scalability of the proposed approach, i.e., whether it works efficiently on large projects with large-scale patches.
A. Setup

1) Subjects: We reused 20 popular Java projects that were chosen by Grund et al. [9] as the subject projects. All of the projects contain rich evolution histories and the number of commits per project varies from 2,506 to 397,649 with a median of 14,748. They also cover a range of domains, including code analysis, unit testing, search engine, distribution framework, and web server. The diversity may help reduce the potential bias in the evaluation.

To further explore the performance of our approach in industrial applications, we selected a closed-source Java project. This is a software refactoring engine developed by one of the world’s leading IT companies, Huawei. It is designed to automatically detect code smells in software applications and to provide tailored solutions for each identified issue. This project contains 298 commits submitted during 2021 and 2023. Therefore, in total we selected 21 subject projects, including 20 open-source projects and one closed-source project.

2) Process: We applied the evaluated approaches independently to the selected subject projects, starting from the latest commit. On each commit, if the evaluated approaches generated identical results (i.e., for all of the changed entities, the matching relationship created by ReMapper and the selected baseline approach was exactly the same) or neither reported any matching relationship (e.g., the commit only modified documents not associated with source code files), we simply dropped the commit and turned to the next one. In contrast, if the evaluated approaches generated inconsistent results (i.e., non-identical results), we collected the commit as a conflict-generating commit, and requested three experienced developers to manually inspect all the matching relations in each conflict-generating commit. Therefore, we could count the number of true positives, the number of false positives, and the number of false negatives for each evaluated approach, thereby computing its precision and recall. All of the participants were required to have Java background. They had a median of 8.5 years of programming experience, 3.5 years working as professional software developers, and 6 years experience with version control systems. In case of inconsistent manual inspection (112 matching relations in total), the participants were requested to discuss together and reach an agreement. Notably, the three developers achieved a high consistency with a Fleiss’ kappa coefficient [26] of 0.85. To control the cost of manual validation, we only selected the most recent 10 conflict-generating commits per open-source project.

For the closed-source project, we requested the original developers of the source code instead of the third-party participants to inspect the matching relations. The developers knew the source code and its evolution well, and thus their inspection could be more accurate and much more cost-effective. To this end, we requested the original developers to validate all (15 in total) conflict-generating commits of the project.

Based on the manual inspection, we computed the performance metrics for the evaluated approaches, i.e., the number of false positives, the number of false negatives, precision, and recall. We also manually analyzed some incorrect matching pairs to gain insights into the reasons of the mistakes.

B. Improving the State of the Art

To measure the performance of our approach in entity matching, we focused on the number of false positives (#FP) and the number of false negatives (#FN). A false positive is a pair of entities \(<e_1, e_2>\) that is reported as matched whereas manual checking marked them as mismatched. In contrast, a false negative is a pair of entities that is manually marked as matched but the matching algorithm does not report it as a matched entity pair. \(\text{MST} = \text{FP} + \text{FN}\) (i.e., the number of false positives plus the number of false negatives) represents how frequently the evaluated approaches make mistakes. The key concern of RQ1 is to what extent ReMapper can reduce the frequency of mistakes (i.e., \(\text{MST}\)). Besides \(\text{MST}\), \#FP, and \#FN, we also computed the precision and recall where precision measures how often the reported pairs were manually confirmed, and recall measures how often matched pairs were retrieved by the evaluated algorithms.

We ignored entities whose scope is confined within a single method, e.g., parameters, variables, and statements, because they are out of the scope of ReMapper. The selected baseline approach ignores “annotation type” and “annotation member”. To favor the baseline, we ignored all entities of these two categories while accounting for false positives and false negatives for the baseline approach (i.e., assuming that it makes no mistakes at all in matching such entities).

Our evaluation results are presented in Fig. 3. The horizontal axis presents the involved projects where the last one (“refactoring-engine”) is the closed-source project whereas others are open-source projects. The vertical axis presents the number of false positives and false negatives as well their sum (i.e., \#FP, \#FN, and \#MST) on each subject project.

From Fig. 3, we observe that compared against the state of the art, ReMapper substantially reduced the frequency of mistakes: The total number of mistakes (i.e., \#MST) was reduced from 579 to 82, with a substantial reduction of 85.8%=(579-82)/579. On average, the number of false positives per project was reduced by 84.1%=(10.7-1.7)/10.7 and the number of false negatives per project was reduced...
by 87%=(16.9-2.2)/16.9. We performed a significance test to validate whether there is a statistically significant difference between the total number of mistakes caused by the two approaches. Our evaluation results (p-value [27]=2.57E-7 and effect size of Cohen’s d [28]=1.96) confirmed that the reduction in #MST was statistically significant. Similar significance tests confirmed that the improvement in precision (p-value=1.87E-6 and Cohen’s d=1.76) and recall (p-value=1.3E-5 and Cohen’s d=1.9) were statistically significant.

We also observe from Fig. 3 that the frequency of mistakes (#MST) was reduced on each of the involved 21 projects (both open-source projects and closed-source projects), which may suggest that ReMapper can stably improve the state of the art regardless of the involved projects. As mentioned in Section III-A1, the 21 subject projects were from different domains. Consequently, the consistent performance on all such diverse projects may suggest that ReMapper works well.

We further investigated their performance on matching different categories of software entities, e.g., “classes”, “interfaces” and “methods”. The evaluation results are presented in Table I. From this table, we observe that ReMapper outperforms the baseline on all of the involved entity types except for “enum” where neither of them made any mistakes.

We also notice that ReMapper resulted in high precision and recall on all of the entity types. The minimal precision (on “field”) was 98.3%, and the minimal recall (on “initializer”) was 88.89%. It may suggest that ReMapper worked well on all entity types. The evaluated approaches reported the highest numbers of mistakes in matching entities of “method”, followed by “field”. It is reasonable because the numbers of involved methods and fields are significantly greater than those of classes, interfaces, enums, and annotation types.

We leverage a real-world example from project flink [29] in Fig. 4 to illustrate how ReMapper avoided some false positives. The method doLogin on Line 73 in Vn was renamed as doLoginAndReturnUGI on Line 109 in Vn+1, and a new method named doLogin on Line 90 in Vn+1 was added. However, the baseline approach mapped method doLogin in Vn by mistake to method doLogin in Vn+1 because their method signatures were identical except for the return types. In contrast, our approach correctly mapped method doLogin in Vn to method doLoginAndReturnUGI in Vn+1 because 1) their method bodies were exactly the same and 2) they were called by the same software entities, i.e., startTokensUpdate and obtainDelegationTokens. It did not map doLogin in Vn to doLogin in Vn+1 although their signatures were highly similar because they were significantly different concerning their references (callers) and method bodies.

We leverage another real-world example from project jmeter [30] in Fig. 5 to illustrate how ReMapper successfully retrieved the pairs (i.e., false negatives) missed by the baseline approach. The baseline approach failed to match method getTestClassNames on Line 1023 in Vn with method getClassShortNames on Line 1015 in Vn+1. The change between these two methods involves a complex refactoring (replace loop with pipeline [31], [32]). The refactoring made the two method bodies dissimilar, which causes the failure of the baseline approach that heavily depends on the statement-level similarity of method bodies. Our approach succeeded because it leverages the reference-based similarity: The two methods are called by exactly the same Java entities.

We also observe from Table I that the highest numbers of mistakes in matching entities of “method”, “interface”, “field”, and “enum” were 88.89%, 99.38%, 99.21%, and 100% respectively. Therefore, we may suggest that ReMapper may be a more robust approach for matching methods and fields.

We further investigate their performance on matching different categories of software entities, e.g., “classes”, “interfaces” and “methods”. The evaluation results are presented in Table I. From this table, we observe that ReMapper outperforms the baseline on all of the involved entity types except for “enum” where neither of them made any mistakes.

We also observe from Fig. 3 that the frequency of mistakes (#MST) was reduced on each of the involved 21 projects (both open-source projects and closed-source projects), which may suggest that ReMapper can stably improve the state of the art regardless of the involved projects. As mentioned in Section III-A1, the 21 subject projects were from different domains. Consequently, the consistent performance on all such diverse projects may suggest that ReMapper works well.

We further investigated their performance on matching different categories of software entities, e.g., “classes”, “interfaces” and “methods”. The evaluation results are presented in Table I. From this table, we observe that ReMapper outperforms the baseline on all of the involved entity types except for “enum” where neither of them made any mistakes.

We also observe from Fig. 3 that the frequency of mistakes (#MST) was reduced on each of the involved 21 projects (both open-source projects and closed-source projects), which may suggest that ReMapper can stably improve the state of the art regardless of the involved projects. As mentioned in Section III-A1, the 21 subject projects were from different domains. Consequently, the consistent performance on all such diverse projects may suggest that ReMapper works well.

We further investigated their performance on matching different categories of software entities, e.g., “classes”, “interfaces” and “methods”. The evaluation results are presented in Table I. From this table, we observe that ReMapper outperforms the baseline on all of the involved entity types except for “enum” where neither of them made any mistakes.

We also observe from Fig. 3 that the frequency of mistakes (#MST) was reduced on each of the involved 21 projects (both open-source projects and closed-source projects), which may suggest that ReMapper can stably improve the state of the art regardless of the involved projects. As mentioned in Section III-A1, the 21 subject projects were from different domains. Consequently, the consistent performance on all such diverse projects may suggest that ReMapper works well.

We further investigated their performance on matching different categories of software entities, e.g., “classes”, “interfaces” and “methods”. The evaluation results are presented in Table I. From this table, we observe that ReMapper outperforms the baseline on all of the involved entity types except for “enum” where neither of them made any mistakes.
TABLE II
IMPACT OF PATCH SIZE

<table>
<thead>
<tr>
<th>Patch Size</th>
<th>#MST</th>
<th>Precision</th>
<th>Recall</th>
<th>#MST</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 [ 4, 38, 72]</td>
<td>0.11</td>
<td>99.51%</td>
<td>99.16%</td>
<td>1.84</td>
<td>90.14%</td>
<td>81.59%</td>
</tr>
<tr>
<td>Q2 [ 73, 118, 156]</td>
<td>0.32</td>
<td>98.01%</td>
<td>98.53%</td>
<td>2.02</td>
<td>94.86%</td>
<td>92.76%</td>
</tr>
<tr>
<td>Q3 [157, 244, 364]</td>
<td>0.37</td>
<td>98.75%</td>
<td>98.86%</td>
<td>3.06</td>
<td>95.61%</td>
<td>91.13%</td>
</tr>
<tr>
<td>Q4 [376, 833, 7154]</td>
<td>0.74</td>
<td>99.03%</td>
<td>98.7%</td>
<td>3.94</td>
<td>94.87%</td>
<td>93.58%</td>
</tr>
</tbody>
</table>

C. Impact of Patch Size

We investigated the impact of patch size on the performance of the evaluated approaches, and the evaluation results are presented in Table II. Notably, to reduce the randomness, we partitioned the involved commits into four equally sized groups according to their patch size: Q1, Q2, Q3, and Q4 where Q1 was composed of the smallest 25% commits and Q4 was composed of the largest 25% commits. The first column of Table II specifies the data groups along with the minimal size, average size, and maximal size of the patches within each data group. Columns 2 to 4 present the performance of ReMapper on the given data group. Columns 5 to 7 present the performance of the baseline approach on the given data group. Notably, the performance metrics are the averages. For example, #MST is the average number of mistakes per commit in the given group.

We observe from Table II that the evaluated approaches made more mistakes with the increase in patch size. This is reasonable because the larger the patches are, the more impacted entities there are and thus the more chance to make mistakes. From Table II, we also observe that ReMapper outperformed the baseline on all categories of the patches, i.e., Q1, Q2, Q3, and Q4. It may suggest that ReMapper can stably improve the state of the art in entity matching regardless of the patch size in the commits. However, as shown in Table II, the patch size has a substantially greater impact on the performance of the baseline approach than that on ReMapper. The average precision of ReMapper varies slightly from 99.51% (Q1) to 99.03% (Q4) whereas the recall varies slightly from 99.16% (Q1) to 98.7% (Q4), suggesting that the patch size had little influence on the performance of ReMapper.

To quantitatively measure the correlation between the patch size and the performance of the evaluated approaches, we compute their Pearson correlation coefficient [33] by taking each commit (patch) as an individual. The computation results suggest that the correlation coefficients between the patch size and our approach’s precision and recall are 0.02 and 0.01, respectively. The correlation coefficients between the patch size and the baseline approach’s precision and recall are 0.11 and 0.1, respectively. It may suggest that the correlation between the patch size and the performance of the evaluated approaches is relatively weak.

D. RQ3: Scalability

The evaluation was conducted on a machine with Intel Core i7-11700 CPU @ 2.50GHz, 16 GB DDR4 memory, 512 GB SSD, Windows 10 OS, and Java 11.0.17 x64 with a maximum of 8GB Java heap memory (i.e., -Xmx8g). We recorded the execution time of ReMapper by using System.nanoTime() Java method [34]. Our evaluation results are presented in Table III. The first column not only specifies the categories of the patches (the same as those in Table II), but also presents the minimal size, average size, and maximal size of the given category. For example, within the first category Q1, the minimal size, average size, and maximal size of the patches are 4, 38, and 72, respectively.

From Table III, we observe that the execution time increased with the increase in patch size. However, ReMapper kept efficient even on the largest patches: The median and average execution time for the largest 25% patches (i.e., Q4) was 2.71 seconds and 4.92 seconds, respectively, with a maximum execution time of 28.3 seconds.
the code [35], [36]. On the other side, the resulting refactoring histories are useful for various tasks, e.g., evaluation of refactoring-related approaches/tools [37]–[40] and refactoring opportunity identification [41], [42].

A few approaches have been proposed to automate the discovery of refactorings [5]–[7], [14], [23], [43], [44]. As the first step, such approaches should match entities between two (successive) versions. Based on the matched entity pairs, they leverage a set of predefined heuristics to discover refactoring operations. For example, if method \( m_1 \) in the old version matches method \( m_2 \) in the new version, and they have different names, such approaches would report a renaming: method \( m_1 \) in the old version has been renamed with the name of \( m_2 \).

B. Baseline and Scope

RefactoringMiner [14] represents the state of the art in automated refactoring discovery. Consequently, in this section, we take it as the baseline approach. Notably, our approach (ReMapper) only matches entities across versions, but it does not propose any heuristics to detect refactoring based on the matched entity pairs. To this end, during the evaluation, we simply reused all heuristics implemented by RefactoringMiner, and note the resulting approach as RefactoringExtractor (ReExtractor for short). As a result, the only difference between ReExtractor and RefactoringMiner is the entity matching algorithms.

It should be noted that the refactoring detection heuristics implemented in RefactoringMiner [14], [45] was not completely separated from the implementation of the entity matching algorithm. Consequently, we cannot simply feed the mapping of ReMapper into its heuristic implementation. In contrast, we had to migrate the heuristics from RefactoringMiner to ours, which was time-consuming. To minimize the cost, the evaluation focused on the following 9 refactoring types only: rename method, rename field, rename class, move class, extract class, extract method, inline method, change return type, and change field type. They were selected because they are popular and frequently applied [46]–[48].

C. Process

We applied the two evaluated approaches to each of the projects selected in Section III-A1. Note that some of the projects have long evolution histories, and the automated discovery of refactorings is time-consuming. Consequently, we only leveraged the latest 1,000 commits if the number of commits in one project was greater than 1,000. If either of the evaluated approaches reported any refactorings from a given commit, we collected the commit as a refactoring-discovering commit. The refactoring-discovering commits containing “extract class” or “inline method” refactorings were first selected until the number of “extract class” and the number of “inline method” reported by the evaluated approaches both exceeded 100 because these two types of refactorings are substantially less popular than the others. Giving them priorities helps to strike a balance among different refactoring types. Other refactoring-discovering commits were selected chronologically so that 20 refactoring-discovering commits per open-source project were selected for the evaluation.

We requested three refactoring experts (different from the participants in Section III-A2) to manually and independently validate the refactorings discovered from the selected commits. All of the participants were required to have Java background and were familiar with software refactoring. They had a median of 6.5 years of programming experience and 4 years experience with software refactoring. In case of inconsistency (52 refactoring operations in total), the participants were requested to discuss together and reach an agreement. The three experts achieved a high consistency with a Fleiss’ kappa coefficient [26] of 0.86. Notably, for the closed-source project, we requested the original developers of the source code instead of the third-party participants to validate all (28 in total) refactoring-discovering commits.

D. Results and Analysis

To measure the performance in refactoring discovery, we counted the number of true positives (#TP), the number of false positives (#FP), and the number of false negatives (#FN). If a refactoring operation was reported by only one of the evaluated approaches and then was manually confirmed by the participants, the refactoring operation was taken as a false negative for the other approach that failed to report it as a potential refactoring. We also computed precision = #TP / (#TP + #FP).

The recall was not computed because we had not manually checked all of the commits, and thus we did not know exactly how many refactoring operations had been missed by the evaluated approaches.

The evaluation results are presented in Table IV. From Table IV, we observe that compared against the baseline approach, ReExtractor substantially reduced the number of false positives and false negatives: On average, the number of false positives was reduced by 72.6%=(84-23)/84 and the number of false negatives was reduced by 76.8%=(142-33)/142. We also notice that the number of true positives was substantially improved by ReExtractor with an improvement of 6.9%=(1696-1587)/1587. Since both approaches were evaluated on the same dataset, they should have equivalent #Positives (i.e., the actually conducted refactorings), and thus the relative improvement in recall equals the relative improvement (6.9%) in #TP because recall = #TP / #Positives.

We notice that ReExtractor achieved comparable improvement on the closed-source project. Compared against the baseline, it improved #TP and precision on this project by 6.8%=(63-59)/59 and 11.4%=(91.3%-81.94%)/81.94%, re-

<table>
<thead>
<tr>
<th>Metrics</th>
<th>ReExtractor</th>
<th>RefactoringMiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>#TP</td>
<td>1696</td>
<td>1587</td>
</tr>
<tr>
<td>#FP</td>
<td>23</td>
<td>84</td>
</tr>
<tr>
<td>#FN</td>
<td>33</td>
<td>142</td>
</tr>
<tr>
<td>Precision</td>
<td>98.66%</td>
<td>94.97%</td>
</tr>
</tbody>
</table>
**TABLE V**

**PERFORMANCE IN DISCOVERING DIFFERENT REFACCTORINGS**

<table>
<thead>
<tr>
<th>Refactoring Type</th>
<th>Approaches</th>
<th>#TP</th>
<th>#FP</th>
<th>#FN</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename Method</td>
<td>ReExtractor</td>
<td>305</td>
<td>6</td>
<td>9</td>
<td>98.07%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>286</td>
<td>18</td>
<td>28</td>
<td>94.08%</td>
</tr>
<tr>
<td>Rename Field</td>
<td>ReExtractor</td>
<td>152</td>
<td>2</td>
<td>2</td>
<td>98.7%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>131</td>
<td>11</td>
<td>23</td>
<td>92.25%</td>
</tr>
<tr>
<td>Rename Class</td>
<td>ReExtractor</td>
<td>135</td>
<td>0</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>127</td>
<td>1</td>
<td>9</td>
<td>99.22%</td>
</tr>
<tr>
<td>Move Class</td>
<td>ReExtractor</td>
<td>190</td>
<td>0</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>187</td>
<td>2</td>
<td>3</td>
<td>98.94%</td>
</tr>
<tr>
<td>Extract Class</td>
<td>ReExtractor</td>
<td>93</td>
<td>3</td>
<td>3</td>
<td>96.88%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>85</td>
<td>7</td>
<td>11</td>
<td>92.39%</td>
</tr>
<tr>
<td>Extract Method</td>
<td>ReExtractor</td>
<td>231</td>
<td>3</td>
<td>10</td>
<td>98.72%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>216</td>
<td>13</td>
<td>25</td>
<td>94.32%</td>
</tr>
<tr>
<td>Inline Method</td>
<td>ReExtractor</td>
<td>88</td>
<td>3</td>
<td>4</td>
<td>96.7%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>86</td>
<td>11</td>
<td>6</td>
<td>88.66%</td>
</tr>
<tr>
<td>Change Return Type</td>
<td>ReExtractor</td>
<td>261</td>
<td>5</td>
<td>3</td>
<td>98.12%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>240</td>
<td>12</td>
<td>24</td>
<td>95.24%</td>
</tr>
<tr>
<td>Change Field Type</td>
<td>ReExtractor</td>
<td>241</td>
<td>2</td>
<td>1</td>
<td>99.18%</td>
</tr>
<tr>
<td></td>
<td>RefactoringMiner</td>
<td>229</td>
<td>8</td>
<td>13</td>
<td>96.62%</td>
</tr>
</tbody>
</table>

respectively. It may suggest that ReExtractor can outperform the baseline on both open-source and closed-source projects.

We further investigated their performance on different refactoring types. The evaluation results are presented in Table V. We observe from Table V that ReExtractor outperformed the baseline approach on all of the involved refactoring types, concerning both recall (#TP) and precision. The maximal improvement in precision was up to 8.04=96.7-88.66 percentage points (for inline method refactorings). We also notice that on all types of refactorings, ReExtractor substantially reduced the number of false positives and false negatives.

V. DISCUSSION

A. Threats to Validity

A threat to internal validity is that the ground truth in the employed benchmark could be inaccurate. To construct the ground truth, we requested three developers to manually match entities involved in the selected commits (from open-source projects). However, the participants were not the original developers, and thus their manual matching could be inaccurate. To mitigate this threat, we requested three experienced developers to independently validate each matching pair, and computed the Fleiss’ kappa coefficient (0.85) to validate the high level of consistency. To further mitigate the threat, we also selected a closed-source project for the evaluation where the original developers constructed the ground truth (both for entity matching and refactoring discovery).

A threat to external validity is the limited testing data. Note that the evaluation involved difficult and time-consuming manual construction of ground truth. Consequently, it is difficult to enlarge the testing dataset. However, evaluation results on small testing dataset may suffer from limited generality. To increase the diversity of the dataset, we selected the most recent 10 conflict-generating commits from each of the 20 subject projects. On one side, the testing data covers diverse applications, increasing the diversity of the dataset. On the other side, it limited the total number of to-be-matched entities, which reduced the cost of manual checking. We did not employ random sampling because it increased the uncertainty in the replication of the evaluation.

A threat to construct validity is that the re-implementation of the refactoring detection heuristics in RefactoringMiner could be inaccurate. Ideally, we should replace the entity matching algorithm in RefactoringMiner and keep other parts (refactoring detection heuristics) intact. However, in RefactoringMiner, the detection heuristics are often combined with the entity matching algorithm, and thus it is difficult to replace the matching algorithm in RefactoringMiner. To this end, we re-implemented the heuristics because such heuristics are simple and easy to implement, and connected the re-implementation to our matching algorithm. However, any bugs in the re-implementation may have biased the evaluation results.

B. Limitations

Currently, ReMapper works on Java programs only. Although the rationale of ReMapper is not language-specific, its implementation depends on the automated parsing of source code, and thus it is language-specific. Besides that, ReMapper depends on the data types of the entities, and thus it may not be easily adapted to dynamically typed languages, e.g., Python and JavaScript where the data types could not be determined via static analysis.

It is often time-consuming to retrieve all references for each of the involved entities within a given commit. That is one of the reasons why existing entity matching algorithms do not exploit references. To minimize the cost, ReMapper takes the following two measures. First, ReMapper only retrieves references for those that could not be matched by qualified names. Most of the involved entities within the changed files could be matched by qualified names, which significantly reduces the cost of retrieving references. Second, ReMapper only considers references whose callers and callees are within the changed files. That is, it does not search unchanged files for references because software entities accessed by such unchanged files must have kept their qualified names intact (otherwise, the qualified name-based access would be obsolete). However, such entities whose qualified names are unchanged should have been matched by qualified names (the first step in Fig. 1), and thus there is no need to retrieve their references (for reference-based matching). Notably, ignoring such unchanged files could significantly reduce the cost of retrieving references because most of the files in the repository are not influenced by the given commit.

VI. RELATED WORK

Entity matching algorithms could be divided into two categories. The first category takes two design models as input and
maps one model into the other, and thus they do not exploit the detailed implementation (e.g., method bodies) of the software entities. UMLDiff, proposed by Xing and Stroulia [3], is the most famous algorithm of this category. Although it takes the source code as input, it actually works on the class diagrams automatically generated according to the source code. Consequently, it maps software entities by mapping one graph into another one where a graph represents a class diagram. Nodes of the graph represent software entities whereas edges represent the relations between entities. Nodes and edges are matched according to their labels’ similarity (i.e., name-based similarity) and similarity in connections (i.e., edges). SiDiff, proposed by Kelter et al. [49] is another algorithm belonging to this category. It matches leaf entities (e.g., methods) according to their types, names, and signatures (like parameters of methods), and then takes a bottom-up tactic to match high-level entities (like classes) by computing their similarity in names, matched sub-entities, common generalization targets, and common packages. Once a high-level entity is matched, it takes a top-down tactic to match its sub-entities. Kim et al. [4] presented an approach to automatically infer changes at or above method headers. They defined a set of rules, i.e., low-level transformations (e.g., replacing return types of methods) on method headers. Two methods match each other if there is a sequence of rules that can transform the head of one method into the head of the other.

The second category takes full advantage of the involved source code. JDiff, proposed by Apiwattanapong et al. [2], [24], is of this category. It first matches classes and interfaces based on their fully qualified names, and then matches methods within matched pairs of classes/interfaces according to method signatures. To figure out the difference between a pair of matched methods, it represents the method bodies as enhanced control flow graphs and matches nodes in the graphs by graph isomorphism. ChangeDistiller, proposed by Fluri et al. [20], is widely-used for fine-grained source code change extraction. Given two ASTs representing the source code before and after the revision, ChangeDistiller discovers a sequence of changes that can turn one tree into another. The key of the algorithm is the similarity-based matching between two AST nodes where the similarity between nodes indicates how many percentages of their descendant nodes are common. GumTree, proposed by Falleri et al. [21] improves ChangeDistiller by removing the assumption that leaf nodes contain a lot of text. It exploits a greedy top-down search algorithm to find the greatest isomorphic subtrees between two ASTs, and exploits a bottom-up algorithm to match software entities based on common (matched) subtrees. RefDiff proposed by Silva et al. [6], [23] leveraged a variation of the TF-IDF weighting scheme [50], and a weighted Jaccard coefficient [51] to compute the similarity between software entities (taken as plain texts), and matched entities according to predefined similarity thresholds. RefactoringMiner proposed by Tsantalis et al. [7], [14] matches entities without any similarity thresholds, which significantly facilitates the usage of the algorithm. Another significant contribution of RefactoringMiner is that it proposed two novel pre-processing techniques (i.e., abstraction and argumentization) to facilitate statement matching. We notice that none of the matching algorithms of this category exploits the references of software entities as our approach does.

Entity matching algorithms are the cornerstone of a variety of software engineering tasks, e.g., refactoring discovery, entity tracking, and change comprehension. Here we present recent applications of the entity matching algorithms introduced in the preceding paragraphs. Xing and Stroulia [52] developed JDEvAn [53] to discover refactoring operations based on the structural changes reported by UMLDiff [3]. RefDiff [6], [23] has also been successfully employed to discover software refactorings. RefactoringMiner [7], [14] is one of the most widely used refactoring discovery tools, and the major reason for its improvement over the state of the art is that it proposed a novel entity matching algorithm as introduced in the preceding paragraph proposed. CodeShovel, proposed by Grund et al. [9], leverages an entity matching algorithm to track evolved methods in the commit history. CodeTracker proposed by Jodavi and Tsantalis [10] utilizes the state-of-the-art entity matching algorithm in RefactoringMiner [7], [14] to track methods in evolution history. CDiff, proposed by Huang et al. [54], is a visual code difference tool, leveraging GumTree [21] to map source code entities across versions.

VII. Conclusions and Future Work

In this paper, we propose an automated iterative approach (called ReMapper) to match software entities between two successive versions. It takes full advantage of the qualified names, the implementations, and the references of software entities. It leverages an iterative matching algorithm to handle the interdependence between entity matching and the computation of reference-based similarity. We evaluate ReMapper on both open-source and closed-source real-world projects. Our evaluation results suggest that ReMapper substantially improved the state of the art in entity matching, reducing the number of mistakes by 85.8%. It also substantially improved the state of the art in automated discovery of software refactorings, improving recall by 6.9% and reducing the number of false positives by 72.6%.

In the future, we would like to adapt the proposed approach to more programming languages, which may significantly improve the usefulness of the proposed approach. We also plan to implement more refactoring detection heuristics so that we can discover more refactorings with ReExtractor.

VIII. Data Availability

The replication package, including the tools and the data, is publicly available [55], [56].

Acknowledgment

The authors would like to thank the anonymous reviewers from ASE 2023 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).


