An Automated Approach to Discovering Software Refactorings by Comparing Successive Versions

Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, He Jiang, and Yanjie Jiang

***Abstract*—Software developers and maintainers frequently conduct software refactorings to improve software quality. Identifying the conducted software refactorings may significantly facilitate the comprehension of software evolution, and thus facilitate software maintenance and evolution. Besides that, the identified refactorings are also valuable for data-driven approaches in software refactoring. To this end, researchers have proposed a few approaches to identifying software refactorings automatically. However, the performance (especially precision) of such approaches deserves substantial improvement. To this end, in this paper, we propose a novel refactoring detection approach, called REEXTRACTOR+. At the heart of REEXTRACTOR+ is a reference-based entity matching algorithm that matches coarse-grained code entities (e.g., classes and methods) between two successive versions, and a context-aware statement matching algorithm that matches statements within a pair of matched methods. We evaluated REEXTRACTOR+ on a benchmark consisting of 400 commits from 20 real-world projects. The evaluation results suggested that REEXTRACTOR+ significantly outperformed the state of the art in refactoring detection, reducing the number of false positives by 57.4% and improving recall by 18.4%. We also evaluated the performance of the proposed matching algorithms that serve as the cornerstone of refactoring detection. The evaluation results suggested that the proposed algorithms excel in matching code entities, substantially reducing the number of mistakes (false positives plus false negatives) by 67% compared to the state-of-the-art approaches.**

***Index Terms*—Software Refactoring, History, Detection, Entity Matching, Software Evolution**

I. INTRODUCTION

SOFTWARE refactoring is to improve software quality by changing the internal structure of a software system whereas its external behaviors are kept intact [1], [2]. The primary goal of software refactoring is to improve the readability and maintainability of software applications [3]–[6]. Various software refactorings, like *move methods*, *extract classes*, and *rename variables*, have been proposed and implemented [7]–[9]. Most of the mainstream IDEs, e.g., Eclipse, IntelliJ IDEA, and Visual Studio, have provided automated or semi-automated tool support for various software refactorings. According to the empirical study conducted by Golubev et al. [10], 40.6% of the developers conduct software refactorings every day.

For various reasons, it is desirable to detect software refactorings conducted on real-world software applications [11]. First, identifying software refactorings can assist developers and maintainers in understanding the evolution of given software applications [5], [12]. On one side, distinguishing refactorings from other changes should explain the changes involved in refactorings well and prevent misinterpretations during code reviews. Such insights are particularly beneficial in large collaborative projects where understanding the rationale behind changes is essential for maintaining consistency and code quality. On the other side, identifying refactorings (like *renamings*) in frameworks and APIs also facilitates (potentially automated) adaptions of their clients [13]. For example, when a public API method is renamed, clients that depend on this component need to adapt their code accordingly. Manually tracking such changes can be error-prone and time-consuming. By identifying refactorings, automated tools can detect and suggest necessary adaptations in client code, thus reducing the potential risks associated with outdated method calls.

Second, the identified refactorings are valuable for data-driven approaches in software refactoring [14]. With the significant advances in data mining and deep learning, data-driven approaches for refactoring recommendations are becoming increasingly prevalent [15]–[17]. However, these approaches heavily rely on large-scale and high-quality training data [18]. The refactoring histories discovered from real-world software applications are thus highly desirable [19]. By building a comprehensive dataset of refactorings, deep learning models can be trained to recognize common patterns, optimize code structure, and reduce technical debt. Such deep learning-based models not only assist developers in making informed decisions about when and where to refactor code, but also provide valuable insights into best practices for code maintenance.

Third, refactoring-related approaches, e.g., code smell detection [20], [21] and refactoring solution recommendation [22], [23], require high-quality testing data for comprehensive evaluation. Refactorings that have actually been conducted in real-world applications, along with the corresponding source code before and after the changes, serve as perfect testing data for such approaches [24]–[26]. By identifying refactorings accurately, developers can evaluate these approaches rigorously in real-world scenarios, validating their effectiveness and uncovering potential limitations. Moreover, these evaluations provide valuable insights into the strengths

B. Liu, H. Liu, and Y. Zhang are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China. E-mail: liubo@bit.edu.cn, liuhui08@bit.edu.cn, yuxiazh@bit.edu.cn

N. Niu is with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA. E-mail: nan.niu@uc.edu

G. Li is with the National Innovation Institute of Defense Technology, Beijing, China. E-mail: liguangjie_er@126.com

H. Jiang is with the School of Software, Dalian University of Technology, Dalian, China. E-mail: jianghe@dlut.edu.cn

Y. Jiang is with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China; She is also with the School of Computer Science, Peking University, Beijing, China. E-mail: jiangyanjiese@gmail.com

Corresponding Authors: Hui Liu and Yanjie Jiang

and weaknesses of these approaches, which can be used to refine them and improve the reliability of the suggestions provided to developers.

However, it is challenging to discover software refactorings from the complex evolution of software applications [27]. Manual discovery of refactorings could be both time-consuming and challenging [28] because changes in non-trivial software applications are often enormous, and such changes (rarely well-documented) are hard to understand [29]. To this end, a few approaches have been proposed to automate the detection of software refactorings [11], [30]–[34]. Such approaches first leverage a matching algorithm to map code entities before refactoring to entities after refactoring. With the result of entity mappings, they leverage a set of heuristic-based rules to identify software refactorings. For example, if method m_a in the old version (i.e., before refactoring) matches method m_b in the new version (i.e., after refactoring), and they are of different names, a refactoring detection approach would report a *rename method* refactoring, suggesting that the method has been renamed from $m_a.name$ to $m_b.name$. Such automated approaches have been widely used to detect refactorings, which makes it possible to build large-scale high-quality refactoring datasets. However, as suggested by the evaluation results in Section III-C, the performance (especially precision) of such approaches deserves substantial improvement.

To this end, in this paper, we present a novel approach (called REEXTRACTOR+) to detect software refactorings. The keys to the approach are two matching algorithms. The first one is a reference-based entity matching algorithm that matches coarse-grained code entities, such as classes and methods, between two successive versions. The entity matching algorithm takes full advantage of the qualified names, implementations, and references of code entities. Among them, the references of an entity are other entities that access it by method call, field access, or class instantiation. The references can effectively distinguish whether two similar code entities are matched or not and thus assist in understanding how code entities evolve. The other key algorithm is a context-aware statement matching algorithm that matches fine-grained code entities (i.e., statements) between the pairs of matched methods. The statement matching algorithm exploits the contexts of the to-be-matched statements (i.e., other statements around them within the innermost enclosing block).

This paper is an expanded version of the conference paper [35] where we proposed a reference-based entity matching algorithm to match coarse-grained code entities between two successive versions. Compared to the conference version, in this paper, we make the following expansion:

- We propose a context-aware statement matching algorithm that exploits the contexts of the to-be-matched statements to facilitate statement matching.
- We present a refactoring detection approach that can detect 28 categories of refactorings covering both high-level and low-level refactorings.
- We construct a new dataset for refactoring detection (called *ref-Dataset*), which increases the dataset size by 104% and is validated by multiple developers.

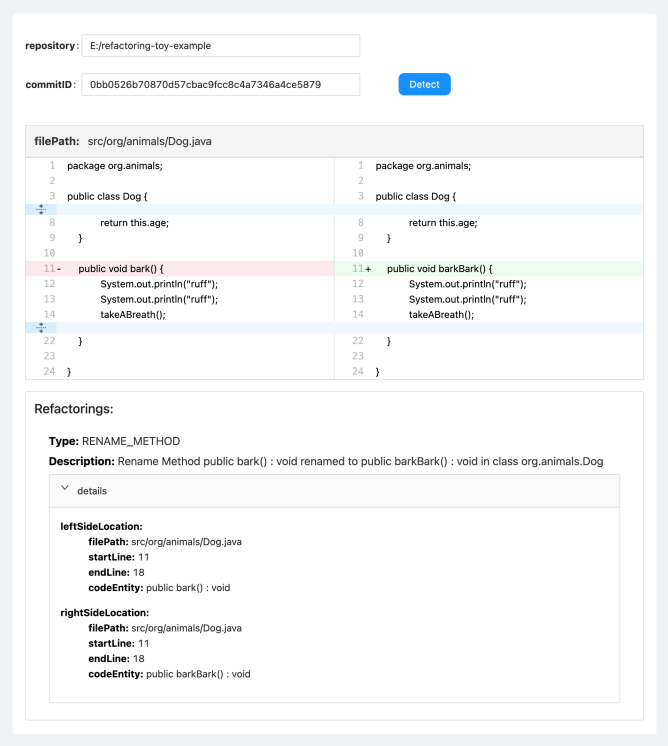

Fig. 1. Visualization of refactorings identified by REEXTRACTOR+

- Our evaluation on the dataset demonstrates the effectiveness of our approach against the baseline approach.
- We develop a web application to visualize the refactorings identified by our approach, including the repository, commit ID, detailed diffs for each commit, and the corresponding refactoring list.

A concrete example is presented in Fig. 1. When a developer inputs a Java project repository and a commit ID into the web application, these parameters are sent to the backend for refactoring detection. Upon receiving the request, the backend is responsible for analyzing the code changes associated with the given repository and commit ID to identify any refactorings that have been conducted. Once the backend has completed the analysis, it generates a comprehensive report detailing the identified refactorings. The result is then sent back to the web application, which presents it to the user clearly and intuitively. The user interface is designed to visually present the identified refactorings, allowing the developer to review and understand the changes effortlessly. The visualization presents detailed information such as the type and description of each refactoring, as well as the affected code entities. The detailed representation provides valuable insights into the changes, facilitating the understanding of the refactorings within the commit.

The rest of the paper is structured as follows. Section II outlines the overall workflow of REEXTRACTOR+ and introduces the matching algorithms that serve as the cornerstone of refactoring detection. Section III evaluates the performance of our approach in comparison to the selected baseline approach. Section IV delves into the threats to validity and limitations of REEXTRACTOR+. Section V provides an overview of related

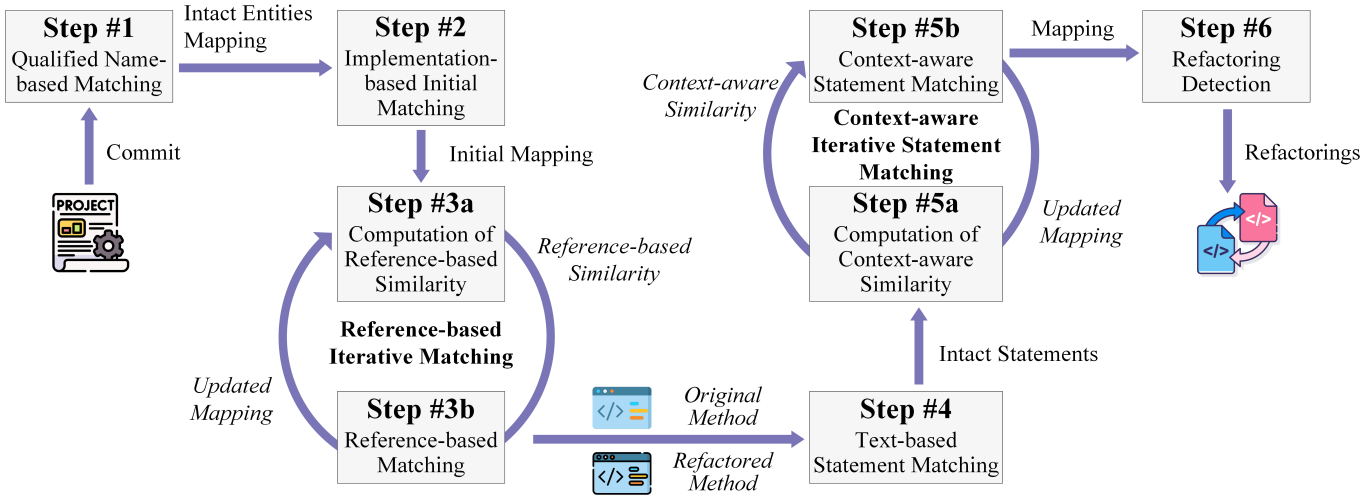

Fig. 2. Overview of REEXTRACTOR+

research on refactoring detection and entity matching. Finally, Section VI summarizes the paper.

II. APPROACH

A. Overview

An overview of the proposed approach is presented in Fig. 2. The approach takes as input a Java project repository and a commit ID, and returns a list of refactorings discovered from the given commit. The overall workflow is explained as follows:

- First, it retrieves all to-be-matched code entities, and distinguishes intact entities (noted as $\mathcal{I}ntactE$) that have not been changed by the given commit. It also leverages a qualified name-based matching algorithm to match entities, and the resulting mapping is noted as $\mathcal{M}$.
- Second, it leverages an implementation-based matching algorithm to initially match entities, and the resulting initial mapping is noted as $\mathcal{M}'$.
- Third, it leverages a reference-based iterative matching algorithm to match (or re-match) entities (excepted for intact entities in $\mathcal{I}ntactE$ and matched entities in $\mathcal{M}$).
- Fourth, for a pair of matched methods/initializers, it leverages a text-based matching algorithm to identify common statements (noted as $\mathcal{I}ntactS$) and match statements that become textually identical after replacements (noted as $\mathcal{S}$).
- Fifth, it leverages a context-aware iterative matching algorithm to match (or re-match) statements (excepted for intact statements in $\mathcal{I}ntactS$ and matched statements in $\mathcal{S}$).
- Sixth, it leverages a set of refactoring heuristics to identify refactorings based on the mappings built in the preceding steps.

Notably, our approach employs a top-down strategy to match code entities between two successive versions. Specifically, we first match coarse-grained code entities (e.g., classes and methods), followed by fine-grained code entities (e.g., statements). On one side, coarse-grained code entities can take full advantage of their references and thus can be matched more accurately than fine-grained code entities. On the other side, statement mappings should be performed within the pairs of already matched methods/initializers to reduce the chances of erroneous matches. Key steps of the proposed approach are explained in detail in the following sections.

B. Qualified Name-based Matching

Given a Java project repository and a commit ID, we retrieve source code files that have been added, removed, or modified by the commit. Notably, entity matching algorithms usually ignore intact files and non-source code files [33], [34]. All newly added files are added to set $\mathcal{AD}$, and all removed files are added to set $\mathcal{RD}$. The modified files are recorded by a set $\mathcal{MD}$. Each item in $\mathcal{MD}$ represents a pair of documents $<d, d'>$ 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 $<d, d'> \in \mathcal{MD}$, we retrieve all code 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, in this step, the proposed approach only matches coarse-grained code entities, i.e., methods, fields, classes, interfaces, enums, @interface (annotation types), initializers, enum constants, and annotation members.

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 and ne are of the same entity type;
- oe and ne share identical fully qualified name; and
- oe 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 Eclipse JDT `ASTNode.toString()` [36] 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 $\mathcal{I}ntactE$, and remove oe from $oldE$ and ne from $newE$.

For renamed and moved documents, all entities within the document (e.g., fields and methods) should be pair-wise matched. To this end, we utilize Eclipse JGit `RenameDetector` [37] to identify such documents and then match the entities within them. Notably, to match such entities, we lose the matching conditions: two entities match if they share identical entity type and identical implementation, regardless of their fully qualified names because renamed and moved documents change the package names and class names.

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 implementations. 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 implementations have been modified by the given commit but their identities (including fully qualified names, method signatures, and data types) are kept intact. If oe and ne match, we add a tuple $<oe, ne>$ to set $\mathcal{M}$, and remove oe from $oldE$ and ne from $newE$.

C. Implementation-based Initial Matching

Entities to be matched are classified into two categories. The first category is composed of code entities that cannot contain other to-be-matched entities. For example, methods belong to this category because the proposed approach does not match code entities within a method (like parameters, variables, and statements) in this step. The other category is composed of code 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. For the sake of simplicity, we call the two categories *atomic entities* and *compound entities*, respectively.

All to-be-matched entities (except for intact entities in $\mathcal{I}ntactE$ 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 such entities by Algorithm 1. For each entity $oe \in 2bmE_n$, we compare it against each 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 two atomic entities are of the same entity type, on Line 5 in Algorithm 1, we compute the implementation-based similarity (i.e., $implSim(oe, ne)$) between them and add a triple $<oe, ne, implSim(oe, ne)>$ to list $\mathcal{L}$ (Line 7) if and only if:

Algorithm 1: Implementation-based Initial Matching

```
   Input  : 2bmE_n and 2bmE_{n+1}
   Output: M'
1  L ← ∅
2  foreach oe ∈ 2bmE_n do
3      foreach ne ∈ 2bmE_{n+1} do
4          if oe.type == ne.type then
5              implSim(oe, ne) = compImplSim(oe, ne)
6              if implSim(oe, ne) ≥ 0.5 then
7                  L.add(<oe, ne, implSim(oe, ne)>)
8              end
9          end
10     end
11 end
12 sortBySim(L)  // in descending order
13 return simBasedMatching(L)
14 Function simBasedMatching(L)
15     M' ← ∅, O ← ∅, N ← ∅
16     for int i = 0; i < L.length; i++ do
17         if L[i].oe ∉ O and L[i].ne ∉ N then
18             M'.add(<L[i].oe, L[i].ne>)
19             O.add(L[i].oe)
20             N.add(L[i].ne)
21         end
22     end
23     return M'
24 end
```

- oe and ne are of the same entity type; and
- $implSim(oe, ne) \geq 0.5$ (i.e., more than half of their AST nodes or sub-entities 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* [38] that is widely used to measure the similarity between two Abstract Syntax Trees (ASTs) [39]–[41]. The implementation-based similarity is computed as follows:

$$implSim(e_1, e_2) = dice(t_1, t_2) = \frac{2 \times |common(nodes_1, nodes_2)|}{|nodes_1| + |nodes_2|}, \quad (1)$$

where t_1 and t_2 are the ASTs associated with entities e_1 and e_2, respectively. $nodes_1$ and $nodes_2$ denote the AST nodes (including all inner nodes and leaf nodes except for the root node) in t_1 and t_2, respectively. Function $common(nodes_1, nodes_2)$ denotes the common nodes between the two lists. Nodes $nd_i \in nodes_1$ and $nd_j \in nodes_2$ represent a common node if they share the same node type (i.e., $nd_i.getNodeType() == nd_j.getNodeType()$) and equal value (i.e., $nd_i.toString() == nd_j.toString()$). An example of initial matching between two atomic entities is presented in Fig. 3. We extend the `ASTVisitor` and override the `preVisit2()` method to traverse the AST in a pre-order fashion and to store all AST nodes in a list. The implementation similarity between the two methods is

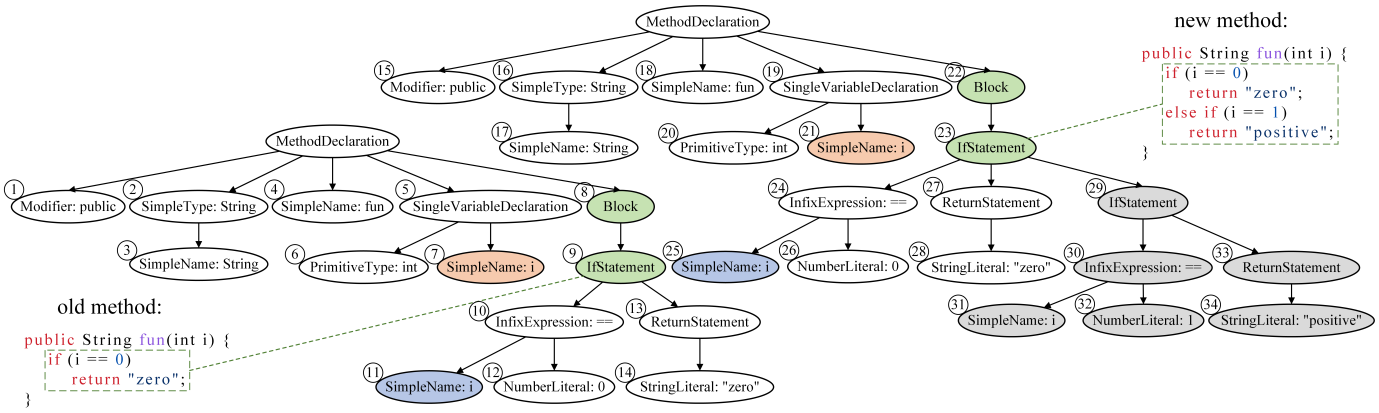

Fig. 3. Two sample Java methods with their corresponding ASTs. The leaf nodes are labeled as `Node Type: Value`. For readability reasons, the internal nodes are labeled simply as `Node Type`. The number on the top-left corner of each label in this figure represents the traversal order.

0.71=2×12/(14+20), where the number of common nodes is 12, that is, all nodes are matched except for nodes ⑧ and ㉓ as well as nodes ⑨ and ㉔, which have different values, and nodes ㉙ to ㉞, which are newly added. For nodes of the same type used in different contexts, e.g., SimpleName, we match such nodes based on their value, type, and their parent nodes' type. As a result, a simple method call name would not match a simple variable name. For complex AST nodes that contain a body, e.g., ForStatement, we treat them as internal nodes within the AST, and their bodies (represented as subtrees) are handled in the same way as other nodes. Notably, we follow GumTree [41] to measure the ratio of common AST nodes and to handle such complex nodes.

We sort $\mathcal{L}$ by implementation-based similarity in descending order (Line 12). Function `simBasedMatching` (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:

$$implSim(e_1, e_2) = \frac{2 \times |common(subE_1, subE_2)|}{|subE_1| + |subE_2|}, \quad (2)$$

where $subE$ denotes all entities contained in e. For example, if e is a class, $subE$ is composed of all methods, fields, and inner types (e.g., inner classes and inner interfaces) declared within e. Function $common(subE_1, subE_2)$ denotes the common entities within $subE_1$ and $subE_2$. The two entities $e_1 \in subE_1$ and $e_2 \in subE_2$ represent a single common entity if and only if e_1 and e_2 have been matched (or temporarily matched), i.e., $<e_1, e_2> \in (\mathcal{I}ntactE \cup \mathcal{M} \cup \mathcal{M}')$. 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.

D. Reference-based Iterative Matching

As suggested by the overview in Fig. 2, reference-based iterative matching is a loop containing two iterative steps, i.e., computation of reference-based similarity and reference-based matching. The key of this step is to retrieve the references to code entities between the two versions. A Reference refers to instances where a code entity e is used by other entities of the software project, e.g., method call, field access, class instantiation, etc. For example, a method call is an entity e (such as a method or a field) calling a method m. In this example, e is considered one of the references of m. To retrieve the references of entities, we build an Entity Reference Graph (ERG) based on the syntax parsing. As discussed in Section II-B, all added, removed, or modified source code files have been retrieved and each file has been parsed into an AST using Eclipse JDT. To illustrate the reference retrieval, we leverage method call as an example to explain how the ERG is built. The overall workflow for reference retrieval is described as follows:

- First, for each code entity *caller* in the added, removed, and modified file, we create a corresponding node in the ERG. After all nodes are created, the graph contains only N nodes without any edges, where N represents the total number of all code entities across these files. Each node uniquely represents a code entity.
- Second, we extend the `ASTVisitor` and override `visit()` method to capture all AST nodes associated with the method call (such as *MethodInvocation*, *SuperMethodInvocation*, *ExpressionMethodReference*, etc.) within each *caller*.
- Third, we leverage `resolveMethodBinding()` method to obtain binding information for each *callee* method, which includes its method signature and the enclosing class to which it belongs. Consequently, a pair of reference relations $<callee, caller>$ can be uniquely identified through binding information, where *caller* is one of the references of *callee*.
- Fourth, if a callee node comes from source code rather than third-party libraries (such as a JAR file), we build the reference relation $<callee, caller>$ into the ERG. That is, a directed edge from callee to caller is added to the graph. Otherwise, no edge is added because these nodes are excluded from the ERG, as will be discussed later in Section IV.

Algorithm 2: Reference-based Iterative Matching

Input : $2bmE_n$, $2bmE_{n+1}$, $\mathcal{I}ntactE$, $\mathcal{M}$, and $\mathcal{M}'$
Output: $\mathcal{M}'$

```
1  while true do
2      L ← ∅
3      foreach e1 ∈ 2bmE_n do
4          foreach e2 ∈ 2bmE_{n+1} do
5              if type-compatible(e1, e2) then
6                  sim(e1, e2) =
                     compSim(e1, e2, IntactE, M, M')
7                  if sim(e1, e2) ≥ 0.5 then
8                      L.add(<e1, e2, sim(e1, e2)>)
9                  end
10             end
11         end
12     end
13     sortBySim(L)  // in descending order
14     M'' ← simBasedMatching(L)
15     if M' == M'' then
16         return M'
17     else
18         M' ← M''
19     end
20 end
```

- Fifth, we repeat the preceding steps until all reference relations for code entities have been resolved, i.e., the ERG is built completely, where each directed edge from code entity e_1 to code entity e_2 represents a reference relation between them.

Notably, if an entity e is referenced by nodes in unchanged source code files in two versions, such reference relations are not retrieved. As a result, it is efficient and rapid to build the ERG only from added, removed, and modified files. Although the ERG may lose some reference relations from unchanged files, it does not impact any performance of our matching algorithm. This is because entities with identical qualified names and signatures are already matched in the qualified name-based matching, and any reference relations involving entities with changed qualified names and/or signatures do not exist in unchanged files. The details will also be discussed later in Section IV.

In the following sections, 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 $\mathcal{M}'$ (i.e., $\mathcal{M}' == \mathcal{M}''$).

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:

$$refSim(e_1, e_2) = \frac{2 \times |common(callers_1, callers_2)|}{|callers_1| + |callers_2|}, \quad (3)$$

where $callers$ denotes code entities that access e. Function $common(callers_1, callers_2)$ denotes entities accessing both e_1 and e_2. Notably, e_i and e_j represent an entity accessing both e_1 and e_2 if and only if $e_i \in callers_1$, $e_j \in callers_2$, and e_i has been matched (or initially matched) to e_j (i.e., $<e_i, e_j> \in (\mathcal{I}ntactE \cup \mathcal{M} \cup \mathcal{M}')$).

The reference-based similarity between e_i and e_j is the average of their reference similarity and implementation-based similarity:

$$sim(e_1, e_2) = \frac{refSim(e_1, e_2) + implSim(e_1, e_2)}{2} \quad (4)$$

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. [33], [34], 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 minimum threshold (0.5), we add the item $<e_1, e_2, sim(e_1, e_2)>$ to list $\mathcal{L}$ (Line 8), suggesting that they have the potential to match each other. The setting of the threshold is inspired by the matching of intact entities in Section II-B. For a pair of intact entities, they naturally match each other even if their references are entirely different, such as requirement changes or bug fixes. In this case, the reference-based similarity defined by Equation 4 between two intact entities is 0.5, and thus we set this value as the minimum threshold for potential matching between any two entities. 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 [42] of their fully qualified names:

$$nameSim(e_1, e_2) = \frac{2 \times |common(2g(qn_1), 2g(qn_2))|}{|2g(qn_1)| + |2g(qn_2))|}, \quad (5)$$

where qn is the fully qualified name of e, and $2g(qn)$ denotes the list of 2-grams within qn. After sorting $\mathcal{L}$, the proposed approach rematches entities by using function `simBasedMatching` on Line 14. Notably, this function has been defined in Algorithm 1.

E. Text-based Matching

Given a pair of methods $<m, m'>$, we retrieve all statements declared in m and m', respectively. All statements declared in m are added into a set (noted as $Statements_M$), and all statements declared in m' are added into another

set (noted as $Statements_{M'}$). Consequently, the text-based matching is to match elements in $Statements_M$ to statements in $Statements_{M'}$ Notably, statements to be matched are classified into two categories: *compound statements* and *atomic statements*. Compound statements are statements that can contain other statements whereas atomic statements cannot contain other compound or atomic statements. For example, for-loop statements belong to compound statements because they can contain other statements within their loop bodies. In contrast, variable declaration statements belong to atomic statements because they generally do not contain any other statements.

1) Pre-Processing: Extract method and inline method refactorings may significantly influence the text-based matching between the two methods. Therefore, the pre-processing of methods is essential for effective text-based statement matching. The extract method refactoring involves moving a subset of statements from method m to a newly added method (i.e., the extracted method m^+), and introducing a method invocation to m^+ in the refactored method m'. The statements within the extracted method should also be included as part of the statement matching between m and m'. That is, all statements in m^+ should be added to $Statements_{M'}$ without changing their original structures and nesting relationships. Notably, a method m^+ is deemed extracted from m (whose revised version is m') if they satisfy all of the following conditions:

- m^+ is a newly added method in the new version;
- m' calls m^+ in the new version;
- One or several statements in m fail to match statements in m' at the granularity of code lines; and
- $\{common(AST(m) - AST(m'), AST(m^+))\} \neq \varnothing$, where $AST(m)$ represents the AST nodes of m.

The inline method refactoring, which involves replacing a method invocation with the body of the invoked method, is pre-processed in a similar but reversed way: All statements in the inlined method are added to $Statements_M$ without changing their original structures and nesting relationships.

2) Statement Matching: We first identify common statements between $Statements_M$ and $Statements_{M'}$ according to the texts of the statements. The identification is composed of two rounds. In the first round, two statements match if and only if they share identical statement type, identical text, and identical nesting depth (within their enclosing methods). Matched statements are added to $\mathcal{I}ntactS$. In the second round, we further match the remaining statements (i.e., statements that have not yet been matched in the first step) by losing the matching conditions: two statements match if they share identical statement type and identical text, regardless of their nesting depth. Such matched statements are also added to $\mathcal{I}ntactS$. Note that multiple matches within a method are ranked according to the lexical similarity of their innermost enclosing blocks.

For the remaining statements in $Statements_M$ and $Statements_{M'}$ that are not matched in the preceding step, we attempt to match them again using heuristics-based rules. Specifically, two statements $os \in Statements_M$ and $rs \in Statements_{M'}$ match if and only if they become textually identical after performing any (or a combination) of the following replacements:

- Renaming (Rule 1): It is to replace old entity names (in os) with new names. Two statements may fail to match due to renaming refactorings. If reversing the renaming refactorings could make statement $os \in Statements_M$ and statement $rs \in Statements_{M'}$ lexically identical, we add the tuple $<os, rs>$ to set $\mathcal{S}$, and remove os from $Statements_M$ and rs from $Statements_{M'}$.
- Argumentization (Rule 2): If rs comes from an extracted method (m^+) and contains parameters of the method (i.e., m^+), we should replace the parameters in rs with their corresponding argument values before lexical matching. For statements derived from inlined methods, similar replacement (i.e., replacing parameters with arguments) should be conducted before lexical comparison as well. Notably, this heuristics has previously been employed by Tsantalis et al. [33], [34].
- Variablization (Rule 3): It is to replace variable names with variables' initialization values. For example, *extract variable* refactorings may extract expressions from statements as a local variable, resulting in the same statements with non-identical text in the two versions. For *inline variable* refactorings, similar replacement should be conducted before lexical comparison as well. By the variablization, we may turn them identical again. Besides, variablization can handle the same statements with non-identical text due to variable renaming.

Notably, if two to-be-matched statements have a higher textual similarity (measured by Levenshtein distance [43]) after a replacement, we perform this replacement. Otherwise, we do not perform this replacement and turn to the next replacement. All matched statements are added to $\mathcal{S}$.

F. Context-aware Iterative Matching

As suggested by the overview in Fig. 2, context-aware iterative matching is a loop containing two iterative steps, i.e., computation of context-aware similarity and context-aware matching. We explain the loop (iteration) first, and then the two key steps within the loop.

1) Iterative Matching: It should be noted that the context-aware similarity depends on the mapping of statements. Meanwhile, the latter depends on the context-aware similarity, as well. To resolve their circular dependency, we employ the iterative matching tactic as specified in Algorithm 3 (Lines 1-9). On the first run of function `ctxBasedMatching` (Line 1), $\mathcal{S}'$ is empty and thus we take $\varnothing$ as the last argument. After the running, $\mathcal{S}'$ contains tuples of temporarily matched statements. We then keep running `ctxBasedMatching` (Line 3) until the mapping $\mathcal{S}'$ becomes stable (i.e., $\mathcal{S}' == \mathcal{S}''$).

2) Computation of Context-aware Similarity: The key to context-aware matching is the computation of context-aware similarity between statements (Line 15 in Algorithm 3). All to-be-matched statements (except for intact statements in $IntactS$ and matched statements in $\mathcal{S}$) from m and m' are noted as $2bmS$ and $2bmS'$, respectively. For a pair of statements $s_1 \in 2bmS$ and $s_2 \in 2bmS'$, their context-aware

Algorithm 3: Context-aware Iterative Matching

Input : $2bmS$, $2bmS'$, $\mathcal{I}ntactS$, and $\mathcal{S}$
Output: $\mathcal{S}'$

1 $\mathcal{S}' \leftarrow$ `ctxBasedMatching`$(2bmS, 2bmS', \mathcal{I}ntactS, \mathcal{S}, \varnothing)$
2 **while** *true* **do**
3 $\quad \mathcal{S}'' \leftarrow$ `ctxBasedMatching`$(2bmS, 2bmS', \mathcal{I}ntactS, \mathcal{S}, \mathcal{S}')$
4 $\quad$ **if** $\mathcal{S}' == \mathcal{S}''$ **then**
5 $\quad\quad$ **return** $\mathcal{S}'$
6 $\quad$ **else**
7 $\quad\quad \mathcal{S}' \leftarrow \mathcal{S}''$
8 $\quad$ **end**
9 **end**
10 **Function** `ctxBasedMatching`$(2bmS, 2bmS', \mathcal{I}ntactS, \mathcal{S}, \mathcal{S}')$
11 $\quad \mathcal{L} \leftarrow \varnothing$
12 $\quad$ **foreach** $s_1 \in 2bmS$ **do**
13 $\quad\quad$ **foreach** $s_2 \in 2bmS'$ **do**
14 $\quad\quad\quad$ **if** `type-compatible`(s_1, s_2) **then**
15 $\quad\quad\quad\quad sim(s_1, s_2) =$ `compSim`$(s_1, s_2, \mathcal{I}ntactS, \mathcal{S}, \mathcal{S}')$
16 $\quad\quad\quad\quad$ **if** $sim(s_1, s_2) \geq 0.5$ **then**
17 $\quad\quad\quad\quad\quad \mathcal{L}$.`add`$(<s_1, s_2, sim(s_1, s_2)>)$
18 $\quad\quad\quad\quad$ **end**
19 $\quad\quad\quad$ **end**
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 $\quad$ `sortBySim`$(\mathcal{L})$ `// in descending order`
23 $\quad \mathcal{S}' \leftarrow \varnothing$, $\mathcal{O} \leftarrow \varnothing$, $\mathcal{R} \leftarrow \varnothing$
24 $\quad$ **for** $int\ i = 0;\ i < \mathcal{L}.length;\ i{+}{+}$ **do**
25 $\quad\quad$ **if** $\mathcal{L}[i].s_1 \notin \mathcal{O}$ and $\mathcal{L}[i].s_2 \notin \mathcal{R}$ **then**
26 $\quad\quad\quad \mathcal{S}'$.`add`$(<\mathcal{L}[i].s_1, \mathcal{L}[i].s_2>)$
27 $\quad\quad\quad \mathcal{O}$.`add`$(\mathcal{L}[i].s_1)$
28 $\quad\quad\quad \mathcal{R}$.`add`$(\mathcal{L}[i].s_2)$
29 $\quad\quad$ **end**
30 $\quad$ **end**
31 $\quad$ **return** $\mathcal{S}'$
32 **end**

similarity depends on their similarity in contexts, texts, and statement types. To this end, we first compute the context similarity between two statements as follows:

$$ctxSim(e_1, e_2) = \frac{2 \times |common(ctx_1, ctx_2)|}{|ctx_1| + |ctx_2|}$$
$$common(ctx_1, ctx_2) = \{<s_i, s_j> \in Matched \mid s_i \in ctx_1 \wedge\ s_j \in ctx_2\}$$
$$Matched = \mathcal{I}ntactS \cup \mathcal{S}, \tag{6}$$

where ctx is the contexts of statement s, including all statements (except for s) within the innermost enclosing block of s. The set $Matched$ indicates all matched statement pairs, including $\mathcal{I}ntactS$ (unchanged common statements matched in Section II-E), $\mathcal{S}$ (matched statements that are textually identical after replacements), and $\mathcal{S}'$ (statements that have been matched so far). Function $common(ctx_1, ctx_2)$ denotes the common statements between the contexts of s_1 and the contexts of s_2. Notably, if the innermost enclosing block of a to-be-matched statement contains only the statement itself (i.e., without any context), the matching of such blocks depends on their parent mappings: if their parent statements are matched, the enclosing blocks are also considered matched. Once the enclosing blocks are confirmed as matched, the standalone statements within the block are matched based solely on text similarity. However, if the blocks are unmatched, the enclosed standalone statements contained within them are deemed unmatched.

Second, we compute the text similarity between the two statements. For two atomic statements, we also leverage the widely-used *dice coefficient* [38] to measure their text similarity $txtSim(s_1, s_2)$ as follows:

$$txtSim(s_1, s_2) = \frac{2 \times |common(nodes_1, nodes_2)|}{|nodes_1| + |nodes_2|}, \tag{7}$$

where t_1 and t_2 are the ASTs associated with statements s_1 and s_2, respectively. $nodes_1$ and $nodes_2$ represent the AST nodes in s_1 and s_2, respectively. Function $common(nodes_1, nodes_2)$ represents the common nodes between the two lists. Nodes $nd_i \in nodes_1$ and $nd_j \in nodes_2$ are considered common if they share the same node type (i.e., $nd_i.getNodeType() == nd_j.getNodeType()$) and equal value (i.e., $nd_i.toString() == nd_j.toString()$). $txtSim(s_1, s_2)$ measures how likely the AST nodes associated with statements s_1 and s_2 are common (overlapping). The more likely they are, the more similar the statements (i.e., s_1 and s_2) are.

For two compound statements, we compute their text similarity as follows:

$$txtSim(s_1, s_2) = \frac{2 \times |common(subS_1, subS_2)|}{|subS_1| + |subS_2|}, \tag{8}$$

where $subS$ represents all statements within s. The common statements of two lists (i.e., $common(subS_1, subS_2)$) should be computed as defined in Equation 6, that is $common(subS_1, subS_2) = \{< s_i, s_j > \in Matched \mid s_i \in subS_1 \wedge\ s_j \in subS_2\}$. The text similarity between two compound statements measures how likely their fine-grained inner statements match. The final context-aware similarity between two statements s_1 and s_2 is the average of their context similarity and text similarity:

$$sim(s_1, s_2) = \frac{ctxSim(s_1, s_2) + txtSim(s_1, s_2)}{2} \tag{9}$$

3) Context-aware Matching: In the preceding paragraphs, we have specified how to compute the similarity between to-be-matched statements, i.e., all statements except for those identified in Section II-E as unchanged common statements and already matched statements. We try to match such statements as specified by function `ctxBasedMatching` in Algorithm 3 (Lines 10-32).

For each to-be-matched statement $os \in 2bmS$, we compute its similarity with each to-be-matched statement $rs \in 2bmS'$ (Lines 12-15) if they are type-compatible (Line 14). Tsantalis

et al. [33], [34] introduced a novel technique called *abstraction* to facilitate the matching of statements by checking whether their expressions are identical or become identical through the replacements of AST nodes. Following them, we also take expression statements [44], return statements [45], variable declaration statements [46], and assignments [47] as type compatible and take loop statements (including for statements [48], enhanced for statements [49], while statements [50], and do statements [51]) as type-compatible.

If their similarity is greater than or equal to a minimum threshold (0.5), we suppose that they have the potential to match, and thus we add the tuple $<os, rs, sim(os, rs)>$ to list $\mathcal{L}$ (Line 18). The setting of the threshold is inspired by the matching of common statements in Section II-E. For a pair of common statements, even if their contexts are entirely different, they are considered to match each other because the second round of text-based matching ignores their nesting depth. For example, developers often restrict the scope of variables by moving them from method-level to block-level, and thus limiting their visibility and lifecycle. In this case, the context-aware similarity defined by Equation 9 between such statements is 0.5 because their contexts are completely different. Thus we set this value as the minimum threshold for potential matching between any two statements. After the enumeration (Lines 12-21), we sort the tuples in $\mathcal{L}$ (Line 22) so that tuples with greater similarities are ranked on the top.

After the ranking of $\mathcal{L}$, we pick up the first tuple in $\mathcal{L}$ and add the tuple as a potential matching to $\mathcal{S}'$ (Line 26). We also add the associated statements in the tuple (i.e., $\mathcal{L}.os$ and $\mathcal{L}.rs$) to empty sets $\mathcal{O}$ and $\mathcal{R}$, respectively. These two sets are employed to record which statements (from the original method m and the improved version m', respectively) have already been temporarily matched. We then go to the next tuple in $\mathcal{L}$, and validate if statements in the tuple have not yet been matched. Only if neither of the statements has been matched, we add the tuple to $\mathcal{S}'$. The iteration ends when all tuples in $\mathcal{L}$ have been checked. Tuples in $\mathcal{S}'$ are returned as the result of the context-aware statement matching.

G. Refactoring Detection

In the previous sections, we match code entities between two successive versions and statements between matched statements respectively. The mappings of entities and statements are recorded with $\mathcal{M}$ and $\mathcal{S}$ respectively. All unmatched entities and statements from the old version (i.e., removed entities and statements) are stored in sets $\mathcal{M}^-$ and $\mathcal{S}^-$, and all unmatched entities and statements from the new version (i.e., added entities and statements) are stored in sets $\mathcal{M}^+$ and $\mathcal{S}^+$. Based on the mappings, REEXTRACTOR+ identifies refactorings according to a set of refactoring heuristics. Note that we are not the first to identify refactorings by heuristics. Existing refactoring detection tools, like REFACTORINGMINER [33], [34], have already defined various heuristic rules to identify different categories of refactorings. Consequently, for these refactorings that have already been supported by existing tools, we simply reuse their heuristics and re-implement them based on our matching relations, as specified in Table I.

For example, given a pair of matched methods $<m_a, m'_a>$ and a newly added method m_b, we could infer an *extract method* refactoring if and only if:

- m_a does not call m_b in the old version;
- m'_a calls m_b in the new version;
- The containers (e.g., *classes* and *interfaces*) of m_a and m'_a are matched; and
- The number of matched statements in m_a and m_b is greater than the number of unmatched statements in m_b

For low-level refactorings, like *extract variable* and *inline variable*, REFACTORINGMINER relies on the syntax-aware replacements of AST nodes to infer such refactorings. In contrast, our approach exploits the contexts of the to-be-matched statements to obtain statement mappings. Therefore, we cannot simply reuse heuristics for such refactorings. To this end, we design two novel refactoring heuristics to infer *extract variable* and *inline variable* refactorings, as presented in the last two rows of Table I. Given a pair of matched statements $<s_a, s'_a>$ and a newly added variable declaration statement s_b, we could infer an *extract variable* refactoring if and only if:

- v is a variable declared in s_b;
- s'_a contains the name of v; and
- s_a contains the initializer of v or $|common(AST(s_a) - AST(s'_a), AST(v.i))| \geq 0.5$, where $AST(v.i)$ represents the AST nodes corresponding to the initializer of v.

III. EVALUATION

In this section, we evaluate the performance and efficiency of the proposed approach REEXTRACTOR+ on real-world projects. We also evaluate the proposed matching algorithms that serve as the cornerstone of REEXTRACTOR+.

A. Research Questions

The evaluation investigates the following research questions:

- **RQ1:** Can REEXTRACTOR+ outperform the state of the art in refactoring detection?
- **RQ2:** Can the proposed matching algorithms improve the state of the art in entity matching?
- **RQ3:** How does the patch size influence the performance of REEXTRACTOR+?
- **RQ4:** Is REEXTRACTOR+ efficient?

RQ1 concerns the performance of the proposed approach and its comparison against the state of the art. To answer RQ1, we take REFACTORINGMINER[1] [34], [52] as the baseline. Notably, RefactoringMiner represents the state of the art in automated refactoring detection, and it is the only one that supports the detection of both high-level refactorings (e.g., *move method*) and low-level refactorings (e.g., *extract variable*). RQ2 concerns whether the proposed matching algorithms can improve the state of the art. To answer RQ2, we compare the proposed matching algorithms against the matching algorithm employed by the state-of-the-art detection approach REFACTORINGMINER. RQ3 concerns whether the

[1]The latest version (release: 3.0.10, commit: 99dfbe4) of REFACTORINGMINER was employed for comparison.

TABLE I
HEURISTICS FOR REFACTORING DETECTION

Refactoring Types	**Heuristics**
Change Method Signature m_a to m_b	$\exists < m_a, m_b > \in \mathcal{M}$ ① $m_a.n \neq m_b.n \Rightarrow$ RENAME METHOD ② $m_a.t \neq m_b.t \Rightarrow$ CHANGE RETURN TYPE ③ $< m_a.c, m_b.c > \notin \mathcal{M} \wedge \neg\texttt{subType}(m_a.c, m_b.c) \wedge \neg\texttt{subType}(m_b.c, m_a.c) \Rightarrow$ MOVE METHOD $\boxed{\texttt{subType}(m_a.c, m_b.c) \Rightarrow \text{PULL UP METHOD}}$ $\boxed{\texttt{subType}(m_b.c, m_a.c) \Rightarrow \text{PUSH DOWN METHOD}}$ $\boxed{m_a.n \neq m_b.n \Rightarrow \text{MOVE \& RENAME METHOD}}$
Extract Method m_b from m_a	$\exists < m_a, m'_a > \in \mathcal{M} \wedge m_b \in \mathcal{M}^+ \wedge < m_a.c, m_b.c > \in \mathcal{M} \wedge \neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m'_a, m_b) \wedge$ $\lvert M \rvert > \lvert U_{T_b} \rvert$
Extract m_b from m_a & Move to $m_b.c$	$\exists < m_a, m'_a > \in \mathcal{M} \wedge m_b \in \mathcal{M}^+ \wedge < m_a.c, m_b.c > \notin \mathcal{M} \wedge \neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m'_a, m_b) \wedge$ $\lvert M \rvert > \lvert U_{T_b} \rvert$
Inline Method m_b to m'_a	$\exists < m_a, m'_a > \in \mathcal{M} \wedge m_b \in \mathcal{M}^- \wedge < m'_a.c, m_b.c > \in \mathcal{M} \wedge \texttt{calls}(m_a, m_b) \wedge \neg\texttt{calls}(m'_a, m_b) \wedge$ $\lvert M \rvert > \lvert U_{T_a} \rvert$
Move m_b to $m'_a.c$ & Inline to m'_a	$\exists < m_a, m'_a > \in \mathcal{M} \wedge m_b \in \mathcal{M}^- \wedge < m'_a.c, m_b.c > \notin \mathcal{M} \wedge \texttt{calls}(m_a, m_b) \wedge \neg\texttt{calls}(m'_a, m_b) \wedge$ $\lvert M \rvert > \lvert U_{T_a} \rvert$
Change Field Signature f_a to f_b	$\exists < f_a, f_b > \in \mathcal{M}$ ① $f_a.n \neq f_b.n \Rightarrow$ RENAME FIELD ② $f_a.t \neq f_b.t \Rightarrow$ CHANGE FIELD TYPE ③ $< f_a.c, f_b.c > \notin \mathcal{M} \wedge \neg\texttt{subType}(f_a.c, f_b.c) \wedge \neg\texttt{subType}(f_b.c, f_a.c) \Rightarrow$ MOVE FIELD $\boxed{\texttt{subType}(f_a.c, f_b.c) \Rightarrow \text{PULL UP FIELD}}$ $\boxed{\texttt{subType}(f_b.c, f_a.c) \Rightarrow \text{PUSH DOWN FIELD}}$ $\boxed{f_a.n \neq f_b.n \Rightarrow \text{MOVE \& RENAME FIELD}}$
Change Class Signature td_a to td_b	$\exists < td_a, td_b > \in \mathcal{M}$ ① $td_a.n \neq td_b.n \Rightarrow$ RENAME CLASS ② $td_a.t \neq td_b.t \Rightarrow$ CHANGE TYPE DECLARATION KIND ③ $< td_a.c, td_b.c > \notin \mathcal{M} \vee td_a.p \neq td_b.p \Rightarrow$ MOVE CLASS $\boxed{td_a.n \neq td_b.n \Rightarrow \text{MOVE \& RENAME CLASS}}$
Extract Class td_b from td_a	$\exists < td_a, td'_a > \in \mathcal{M} \wedge td_b \in \mathcal{M}^+ \wedge \neg\texttt{subType}(td'_a, td_b) \wedge \neg\texttt{subType}(td_b, td'_a) \wedge$ $\exists\,\texttt{move}(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\,\texttt{move}(f_a, f_b) \mid f_a \in td_a \wedge f_b \in td_b$
Extract Superclass td_b from td_a	$\exists < td_a, td'_a > \in \mathcal{M} \wedge td_b \in \mathcal{M}^+ \wedge \texttt{subType}(td'_a, td_b) \wedge \exists\,\texttt{pullUp}(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \vee$ $\exists\,\texttt{pullUp}(f_a, f_b) \mid f_a \in td_a \wedge f_b \in td_b$ $\boxed{\texttt{isInterface}(td_b) \Rightarrow \text{EXTRACT INTERFACE}}$
Extract Subclass td_b from td_a	$\exists < td_a, td'_a > \in \mathcal{M} \wedge td_b \in \mathcal{M}^+ \wedge \texttt{subType}(td_b, td'_a) \wedge \exists\,\texttt{pushDown}(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \vee$ $\exists\,\texttt{pushDown}(f_a, f_b) \mid f_a \in td_a \wedge f_b \in td_b$
Change Variable Signature v_a to v_b	$\exists < s_a, s_b > \in \mathcal{S}, v_a = \texttt{variableDecl}(s_a) \wedge v_b = \texttt{variableDecl}(s_b)$ ① $v_a.n \neq v_b.n \Rightarrow$ RENAME VARIABLE ② $v_a.t \neq v_b.t \Rightarrow$ CHANGE VARIABLE TYPE
Extract Variable v from s_a	$\exists < s_a, s'_a > \in \mathcal{S} \wedge s_b \in \mathcal{S}^+ \wedge (\texttt{contains}(s_a, v.i) \vee \lvert M \rvert > \lvert U_{T_{v.i}} \rvert) \wedge \texttt{contains}(s'_a, v.n) \mid v = \texttt{var}(s_b)$
Inline Variable v to s'_a	$\exists < s_a, s'_a > \in \mathcal{S} \wedge s_b \in \mathcal{S}^- \wedge (\texttt{contains}(s'_a, v.i) \vee \lvert M \rvert > \lvert U_{T_{v.i}} \rvert) \wedge \texttt{contains}(s_a, v.n) \mid v = \texttt{var}(s_b)$

$\texttt{subType}(c_a, c_b)$ returns true if c_a is a direct or indirect subclass of c_b or if it implements interface c_b $\texttt{calls}(m_a, m_b)$ returns true if method m_a calls method m_b $\texttt{move}(m_a, m_b)$ returns true if method m_a moves to method m_b $\texttt{move}(f_a, f_b)$ returns true if method f_a moves to method f_b $\texttt{isInterface}(td)$ return true if td is an interface $\texttt{contains}(s, v.n)$ returns true is statement s contains the name n of variable v $\texttt{var}(s)$ return the variable declaration for statement s

performance of REEXTRACTOR+ may be influenced by the patch size. It is likely that the larger the patch (change) is, the more challenging the refactoring detection will be. By answering RQ3, we may validate this hypothesis. RQ4 concerns the efficiency of the proposed approach, i.e., whether it works efficiently on large projects with large-scale patches.

B. Setup

1) Subjects: We evaluated our approach and the baseline approach using two benchmarks. The first one is the refactoring benchmark constructed by Tstantalis et al. [53]. However, the majority of the commits in this benchmark are from 2015. Given the significant evolution in software development practices over the past decade, it is necessary to validate our approach against more recent commits. To this end, we constructed a new refactoring benchmark tailored to this purpose. We reused 20 popular Java projects that were chosen by Grund et al. [54] as the subject projects. All of the projects contain rich evolution histories and the number of commits per project varies from 2,509 to 439,041 with a median of 16,680 and a total of 844,605. 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.

2) Process: First, the refactoring benchmark was constructed by Tsantalis et al. using triangulation between multiple refactoring tools and human experts [33], [34]. However, the refactoring tool selected for this benchmark did not include our approach. As a result, this benchmark may be biased against our approach because it might overlook some commits where our approach could identify refactorings but the baseline approach failed to identify. To this end, we followed their steps and applied our approach and the baseline approach independently to the projects and commits in this benchmark. We requested three refactoring experts to independently and manually validate all identified refactorings. All of the participants were required to have Java background. They had a median of 7.5 years of programming experience and 5 years of

experience with software refactorings. The validation process was time-consuming and labor-intensive, involving 3 experts for a period of 2.5 months (i.e., 7.5 person-months).

Second, we independently applied the two evaluated approaches to the selected 20 projects, starting from the latest commit. On each commit, if the evaluated approaches generated identical results (i.e., the refactorings identified by REEXTRACTOR+ and REFACTORINGMINER were exactly the same), we simply dropped the commit and turned to the next one because the identical results between the evaluated approaches indicate that their true positives, false positives, and false negatives are also the exactly same, resulting in no statistical difference in the results. In contrast, if the evaluated approaches generated non-identical results, we collected the commit as a refactoring-conflict commit, and requested the three refactoring experts to independently and manually check refactorings discovered from the selected commit. In case of inconsistent manual checking, the participants were requested to discuss together and reach an agreement. Notably, the three developers achieved a high consistency with a Fleiss' kappa coefficient [55] of 0.82. To control the cost of manual validation, we only selected the most recent 20 refactoring-conflict commits per open-source project chronologically. Notably, we did not employ random sampling because it increased the uncertainty (i.e., possible bias in evaluation results) in the replication of the evaluation. In total, we selected 400 commits (20 projects $\times$ 20 commits) for our evaluation[2]. The size of the sample guaranteed a confidence level of over 95% and a margin of error of 5% [56], and thus we believed that the conclusions drawn on the selected commits can be generalized to the entire population (844,605 commits from 20 projects) and are statistically meaningful. The validation process was time-consuming and labor-intensive, involving 3 experts for a period of 1.5 months (i.e., 4.5 person-months).

We also applied the proposed matching algorithms and those in the baseline approach independently to the selected projects. However, in a commit, the number of matching relations is much greater than the number of refactorings. To control the cost of manual validation, we only selected the most recent 3 commits per project chronologically. We requested three experienced developers (different from the participants in the previous step) to independently and manually check all matching relations in each commit. All of the participants were required to have Java background. They had a median of 9.5 years of programming experience, 4.5 years working as professional software developers, and 7 years of experience with version control systems. In case of inconsistent manual checking, the participants were requested to discuss together and reach an agreement. Notably, the three developers achieved a high consistency with a Fleiss' kappa coefficient [55] of 0.87.

C. RQ1: Performance in Refactoring Detection

To measure the performance of REEXTRACTOR+ in refactoring detection, 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 computed precision=#TP/(#TP+#FP) and recall=#TP/(#TP+#FN), where precision measures how often the identified refactorings were manually confirmed and recall measures how often refactorings were discovered by the evaluated approaches. The key concern of RQ1 is to what extent REEXTRACTOR+ can reduce the frequency of false positives and false negatives. Besides precision, we also computed Error Rate=#FP/(#FP+#TP) and Miss Rate=#FN/(#FN+#TP), which represent the frequency of false positives and false negatives, respectively.

TABLE II
PERFORMANCE IN REFACTORING DETECTION ON REFACTORINGMINER'S BENCHMARK

Metrics	REEXTRACTOR+	REFACTORINGMINER
#TP	5489	5514
#FP	273	197
#FN	1062	1037
Precision	95.26%	96.55%
Recall	83.79%	84.17%
Error Rate	4.74%	3.45%
Miss Rate	16.21%	15.83%

Our evaluation results on the refactoring benchmark of REFACTORINGMINER are presented in Table II. We observe from Table II that the performance of our approach is comparable to that of the baseline approach. Specifically, compared to REFACTORINGMINER, REEXTRACTOR+ demonstrates only a 1.29%=(96.55%-95.26%) reduction in precision, along with an almost negligible recall reduction of merely 0.38%=(84.17%-83.79%). We can observe from this table that the evaluated approaches exhibit relatively low recall because these approaches rely on distinct detection methods, each of which has inherent strengths and limitations in identifying refactorings. By integrating these two approaches, a more comprehensive refactoring detection can be achieved, leading to improved recall. Therefore, it is essential to propose a novel refactoring detection approach. Notably, the evaluation results differ from those reported by REFACTORINGMINER because we introduced a new refactoring detection tool and re-validated the refactorings identified by the two evaluated approaches.

We leverage a real-world example from project Netty [57], as depicted in Fig. 4, to illustrate a refactoring missed by REFACTORINGMINER. In this example, the developer conducted two *inline method* refactorings whereas REFACTORINGMINER only identified one of them: *inline principal to getLocationPrincipal*. Consequently, REFACTORINGMINER resulted in a false negative. In contrast, REEXTRACTOR+ could successfully identify these two refactorings. We leverage another real-world example from project Zuul [58], as depicted in Fig. 5, to illustrate an incorrectly validated refactoring REFACTORINGMINER. In this example, the developer renamed the method *getLocations* on Line 193 in V_n to method *getDirectories* on Line 197 in V_{n+1}. REEXTRACTOR+ suc-

[2]The commits analyzed in this study were newly collected from the selected projects on March 28, 2024, rather than reusing the commit dataset provided by Grund et al. [54]

```
1444 1444     public Principal getPeerPrincipal() throws SSLPeerUnverifiedException {
1445 1445         Certificate[] peer = getPeerCertificates();
1446 1446         if (peer == null || peer.length == 0) {
1447 1447             return null;
1448 1448         }
1449      -      return principal(peer);
     1449 +      return ((java.security.cert.X509Certificate) peer[0]).getSubjectX500Principal();
1450 1450     }
1453 1453     public Principal getLocalPrincipal() throws SSLPeerUnverifiedException {
1454 1454         Certificate[] local = getLocalCertificates();
1455 1455         if (local == null || local.length == 0) {
1456 1456             return null;
1457 1457         }
1458      -      return principal(local);
     1458 +      return ((java.security.cert.X509Certificate) local[0]).getIssuerX500Principal();
1459 1459     }
1461      -  public Principal principal(Certificate[] certs) {
1462      -      return ((java.security.cert.X509Certificate) certs[0].getIssuerX500Principal();
1463      -  }
```

V_n (ID:bb17071) V_{n+1} (ID:303cb53)

Fig. 4. Refactoring Missed by the Selected Baseline

```
135 135     List<File> getFiles() {
136 136         List<File> list = new ArrayList<File>();
137     -       for (String sDirectory : config.getLocations()) {
    138 +       for (String sDirectory : config.getDirectories()) {
138 139             if (sDirectory != null) {
... ...                 ......
147 148             }
148 149         }
149 150         return list;
150 151     }

177 179     public static class FilterFileManagerConfig
178 180     {
179     -       private String[] locations;
    181 +       private String[] directories;
    182 +       private String[] classNames;

193     -       public String[] getLocations() {
194     -           return locations;
    197 +       public String[] getDirectories() {
    198 +           return directories;
    199 +       }
    200 +       public String[] getClassNames()
    201 +       {
    202 +           return classNames;
195 203         }
202 210     }
```

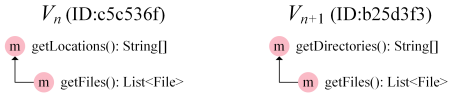

Fig. 5. Refactoring Incorrectly Validated by the Selected Baseline

cessfully identified this refactoring because they were called by the same method, i.e., *getFiles* on Lines 137 and 138. However, REFACTORINGMINER mistakenly identified that method *getLocations* in V_n was renamed to method *getClassNames* in V_{n+1}, and this misidentification was incorrectly marked as a true positive by the validators in the benchmark.

Our evaluation results on the new benchmark (comprising 400 commits from 20 open-source projects) are presented in Table III, all of which were manually validated. From this table, we make the following observations:

- REEXTRACTOR+ substantially reduced the number of false positives and false negatives: In total, the number of false positives was reduced by 57.4%=(408-174)/408 and the number of false negatives was reduced by 54%=(869-400)/869.
- The number of true positives was substantially improved by REEXTRACTOR+ with an improvement of 18.4%=(3014-2545)/2545. The relative improvement in precision and recall are 9.7%=(94.54%-86.18%)/86.18% and 18.4%=(88.28%-74.55%)/74.55%, espectively.

TABLE III
PERFORMANCE IN REFACTORING DETECTION ON NEW BENCHMARK

Metrics	REEXTRACTOR+	REFACTORINGMINER
#TP	3014	2545
#FP	174	408
#FN	400	869
Precision	94.54%	86.18%
Recall	88.28%	74.55%
Error Rate	5.46%	13.82%
Miss Rate	11.72%	25.45%

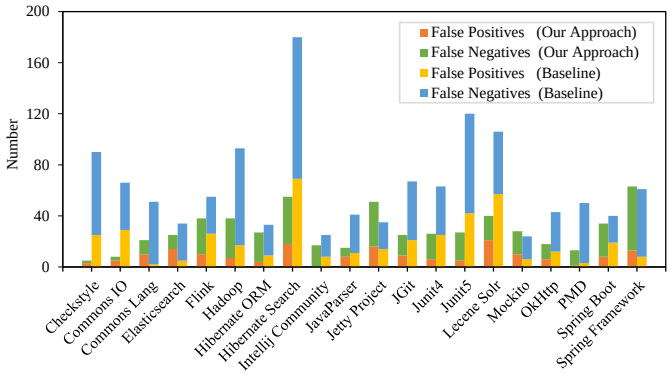

Fig. 6. Number of False Positives and False Negatives per Project

From the analysis of Table II and Table III, it is evident that the performance of the baseline approach exhibits significant variability across the two benchmarks. This discrepancy may be due to the textual differences in the commits between the two benchmarks. Upon further analysis, we identified that the most common mistakes in the baseline approach were caused by incorrect class matching, which typically occurred when classes with highly similar implementations were renamed and/or moved across multiple files. In our benchmark, 4.95%=19/400 of such commits led to matching mistakes in the baseline approach, whereas the benchmark used by `RefactoringMiner` contained only one commit of this nature. Compared against the baseline approach, which relies solely on the implementations of the classes, our approach incorporates reference relations among classes, facilitating more accurate class matching. This integration enables our approach to better handle cases where classes with highly similar implementations are renamed and/or moved, thereby reducing the likelihood of matching mistakes and improving overall accuracy.

Fig. 6 illustrates the comparison on individual projects. The horizontal axis presents the involved projects. The vertical axis presents the number of false positives and false negatives as well as their sum on each subject project. From this figure, we observe that on most cases (17 out of the 20 subject projects) REEXTRACTOR+ reduces the number of mistakes (i.e., the number of false positives plus the number of false negatives).

We employed the Wilcoxon signed-rank test [59] and used Cliff's Delta (d) as the effect size [60] to assess the statistical significance of the observed reduction in false positives and

TABLE IV
PERFORMANCE PER REFACTORING TYPE

Refactoring Type	REEXTRACTOR+			REFACTORINGMINER			Δ *Improvement*		
	#TP	Precision	Recall	#TP	Precision	Recall	#TP	Precision	Recall
Rename Class	226	98.26%	98.69%	152	83.06%	66.38%	*74 (48.7%)*	*15.2% (18.3%)*	*32.31% (48.7%)*
Rename Method	259	92.17%	88.7%	261	86.42%	89.38%	*-2 (-0.8%)*	*5.75% (6.7%)*	*-0.68% (-0.8%)*
Rename Field	103	96.26%	88.03%	96	88.89%	82.05%	*7 (7.3%)*	*7.37% (8.3%)*	*5.98% (7.3%)*
Rename Variable	259	89%	84.92%	236	84.89%	77.38%	*23 (9.7%)*	*4.11% (4.8%)*	*7.54% (9.7%)*
Move Class	140	100%	98.59%	104	65.41%	73.24%	*36 (34.6%)*	*34.59% (52.9%)*	*25.35% (34.6%)*
Move & Rename Class	21	95.45%	70%	28	65.12%	93.33%	*-7 (-25%)*	*30.33% (46.6%)*	*-23.33% (-25%)*
Move Method	267	94.68%	97.8%	248	94.66%	90.84%	*19 (7.7%)*	*0.02% (0%)*	*6.96% (7.7%)*
Move & Rename Method	20	71.43%	71.43%	17	77.27%	60.71%	*3 (17.6%)*	*-5.84% (-7.6%)*	*10.72% (17.7%)*
Move Field	45	91.84%	86.54%	31	91.18%	59.62%	*14 (45.2%)*	*0.66% (0.7%)*	*26.92% (45.2%)*
Move & Rename Field	8	100%	88.89%	2	100%	22.22%	*6 (300%)*	*0% (0%)*	*66.67% (300%)*
Pull Up Method	45	100%	90%	30	69.77%	60%	*15 (50%)*	*30.23% (43.3%)*	*30% (50%)*
Push Down Method	23	100%	95.83%	13	92.86%	54.17%	*10 (76.9%)*	*7.14% (7.7%)*	*41.66% (76.9%)*
Pull Up Field	13	100%	92.86%	11	100%	78.57%	*2 (18.2%)*	*0% (0%)*	*14.29% (18.2%)*
Push Down Field	6	100%	100%	5	100%	83.33%	*1 (20%)*	*0% (0%)*	*16.67% (20%)*
Extract Method	304	98.06%	79.79%	270	92.78%	70.87%	*34 (12.6%)*	*5.28% (5.7%)*	*8.92% (12.6%)*
Extract & Move Method	106	93.81%	89.08%	53	70.67%	44.54%	*53 (100%)*	*23.14% (32.7%)*	*44.54% (100%)*
Extract Class	56	90.32%	94.92%	32	94.12%	54.24%	*24 (75%)*	*-3.8% (-4%)*	*40.68% (75%)*
Extract Superclass	6	85.71%	100%	6	75%	100%	*0 (0%)*	*10.71% (14.3%)*	*0% (0%)*
Extract Subclass	9	100%	100%	3	100%	33.33%	*6 (200%)*	*0% (0%)*	*66.67% (200%)*
Extract Interface	5	100%	83.33%	4	44.44%	66.67%	*1 (25%)*	*55.56% (125%)*	*16.66% (25%)*
Extract Variable	233	93.2%	82.04%	174	91.1%	61.27%	*59 (33.9%)*	*2.1% (2.3%)*	*20.77% (33.9%)*
Inline Method	49	92.45%	66.22%	50	78.13%	67.57%	*-1 (-2%)*	*14.32% (18.3%)*	*-1.35% (-2%)*
Move & Inline Method	55	96.49%	98.21%	39	84.78%	69.64%	*16 (41%)*	*11.71% (13.8%)*	*28.57% (41%)*
Inline Variable	95	93.14%	81.9%	48	80%	41.38%	*47 (97.9%)*	*13.14% (16.4%)*	*40.52% (97.9%)*
Change Type Declaration	1	100%	25%	4	100%	100%	*-3 (-75%)*	*0% (0%)*	*-75% (-75%)*
Change Return Type	169	93.89%	93.89%	146	93.59%	81.11%	*23 (15.8%)*	*0.3% (0.3%)*	*12.78% (15.8%)*
Change Field Type	153	99.35%	90.53%	155	91.72%	91.72%	*-2 (-1.3%)*	*7.63% (8.3%)*	*-1.19% (-1.3%)*
Change Variable Type	338	93.89%	88.95%	327	86.74%	86.05%	*11 (3.4%)*	*7.15% (8.2%)*	*2.9% (3.4%)*
Total	3014	94.54%	88.28%	2545	86.18%	74.55%	*469 (18.4%)*	*8.36% (9.7%)*	*13.73% (18.4%)*

false negatives for the evaluated approaches by taking each commit as an individual. The evaluation results confirmed that the reduction in #FP (p-value [61]=2.6E-5 $\ll 0.05$ and Cliff's $|d|$=0.11) and #FN (p-value=2.93E-9 and Cliff's $|d|$=0.24) was statistically significant. Similarly, the significant test confirmed that the improvement in precision (p-value=5.22E-6 and Cliff's $|d|$=0.17) and recall (p-value=1.37E-7 and Cliff's $|d|$=0.23) was statistically significant.

We further investigated their performance on different refactoring types. The evaluation results are presented in Table IV. The last three columns present to what extent REEXTRACTOR+ outperformed the baseline approach. Note that numbers outside brackets are absolute improvement whereas numbers within brackets are relative improvement. We observe from Table IV that REEXTRACTOR+ outperformed the baseline approach or achieved comparable performance on three-quarters of all involved refactoring types, improving both precision and recall while identifying more true positives.

We leverage a real-world example from project JavaParser [62], as depicted in Fig. 7, to illustrate how REEXTRACTOR+ avoided some false positives. According to the commit message, the developers conducted two rename-related refactorings in this commit, namely, the renaming of class *ConfigurationOption* to *DefaultConfigurationOption* and the renaming of interface *ConfigurableOption* to *ConfigurationOption*. Unfortunately, REFACTORINGMINER erroneously identified a change in type declaration kind for class *ConfigurationOption* (from class to interface) and a subclass

Fig. 7. An example of False Positives from Commit e405cee in JavaParser

extraction from class *ConfigurationOption* to class *DefaultConfigurationOption*. This misidentification occurred during the matching phase, where class *ConfigurationOption* in V_n was mistakenly matched to interface *ConfigurationOption* in V_{n+1}. In contrast, our approach successfully avoided these mistakes because the proposed matching algorithm accurately matched the developers' actual intentions.

Fig. 8 illustrates how REEXTRACTOR+ exploits references to accurately identify and match the renaming of both classes and interfaces. Specifically, interface *ConfigurableOption* on Line 1948 (as a type parameter) and class *ConfigurationOption* on Line 1949 (as a class instantiation) were accessed by method *getOption* in V_n. Similarly, in V_{n+1}, interface *ConfigurationOption* on Line 1948 (as a type parameter) and class *DefaultConfigurationOption* on Line 1949 (as a class instantiation) were accessed by the same method. Consequently, based on their references, REEXTRACTOR+ successfully matched these two interfaces as well as these two classes, effectively

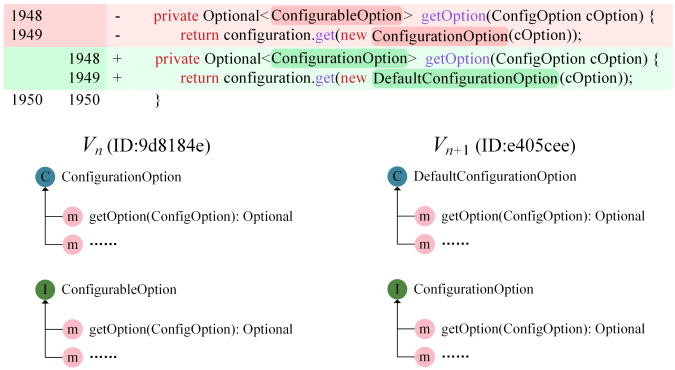

Fig. 8. False Positive Avoided by REEXTRACTOR+

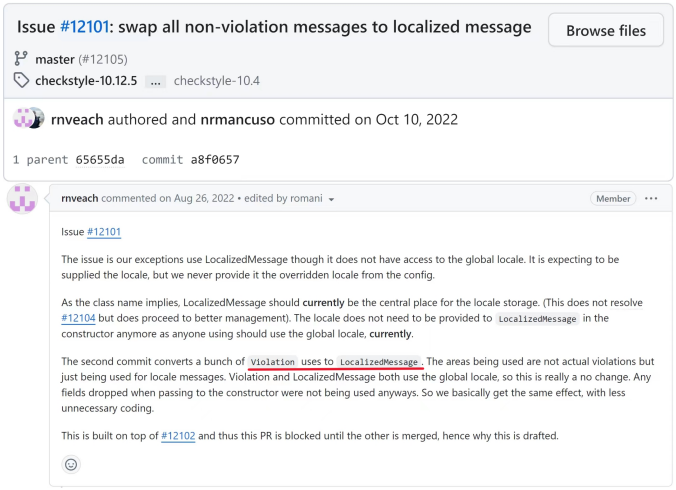

Fig. 9. An example of False Negatives from Commit a8f0657 in Checkstyle

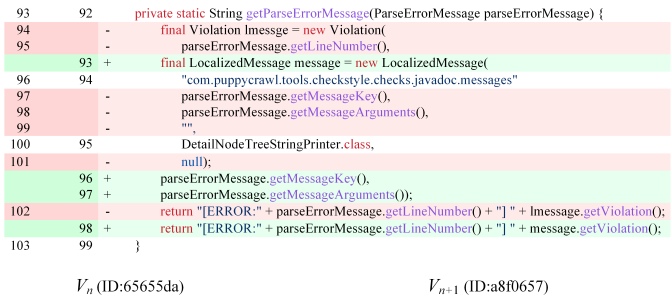

Fig. 10. False Negative Avoided by REEXTRACTOR+

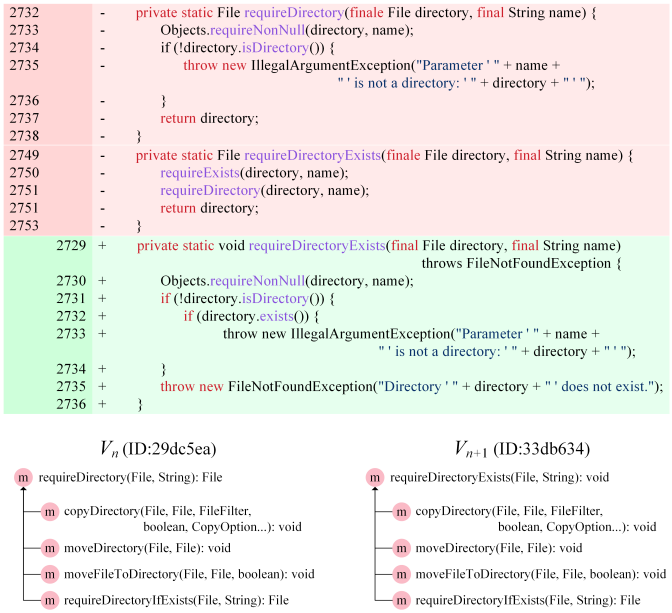

Fig. 11. False Positive And False Negative Due to References

avoiding false positives generated by the baseline approach.

We leverage another real-world example from project Checkstyle [63], as depicted in Fig 9, to illustrate how REEXTRACTOR+ effectively avoids two instances of false negatives, i.e., change variable type *Violation* to *LocalizedMessage* and rename variable *lmessage* to *message*. Fig. 10 depicts the differences between the matched methods. In this example, the developers transition numerous usages of *Violation* into *LocalizedMessage*. Unfortunately, the baseline approach only matched the return statement on Line 102 in V_n and the return statement on Line 98 in V_{n+1}. It failed, however, to identify and match variable declaration *lmessage : Violation* spanning Lines 94-101 in V_n and variable declaration *message : LocalizedMessage* spanning Lines 93-97 in V_{n+1}. The mismatching arose because these two variable declaration statements cannot be rendered textually consistent through AST node replacements. Consequently, the baseline approach resulted in two false negatives. In contrast, our approach successfully matched not only the two return statements but also the two variable declaration statements. Based on the mappings, REEXTRACTOR+ inferred that *change variable type* and *rename variable* refactorings accurately.

We also notice that on all types of refactorings, REEXTRACTOR+ reduced the number of false positives and false negatives. Among them, the maximum reduction including both false positives and false negatives is *move class* refactorings (i.e., 97.8%=(93-2)/93), followed by *rename class* refactorings (i.e., 93.5%=(108-7)/108). It is reasonable because the larger code entities (e.g., classes and interfaces) encapsulate broader reference relations. We could infer that the more references involved in the code entities, the lower the likelihood of matching mistakes. Consequently, this is likely to lead to diminished performance in the detection of refactoring.

Although exploiting references improves the overall performance of the proposed approach, the references occasionally result in false positives and false negatives. A typical example from project Commons IO [64] is presented in Fig. 11. In this example, method *requireDirectory* on Line 2732 in V_n and method *requireDirectoryExists* on Line 2729 in V_{n+1} were incorrectly matched by our approach. This misidentification occurred because the two methods were called by the same code entities. As a result, REEXTRACTOR+ reported two false positives: A *rename method* refactoring and a *change return type* refactoring based on the incorrect matching. In contrast, REFACTORINGMINER matched method *requieDirectoryExists* on Line 2749 in V_n and method *requireDirectoryExists* on Line 2729 in V_{n+1} because their signatures were of the same except for the return types. As a result, REFACTORINGMINER

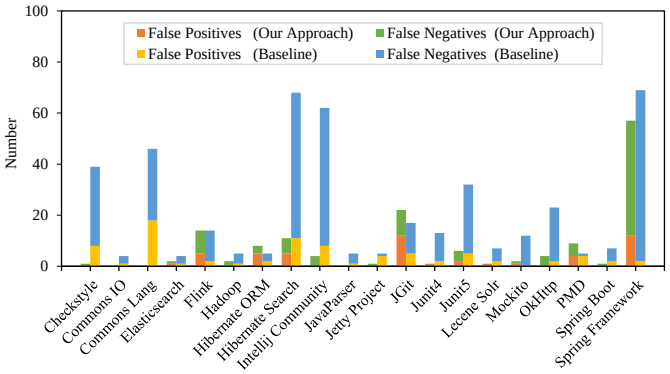

Fig. 12. Number of Mistakes per Project

reported an *inline mthod* refactoring: Inline method *requireDirectory* to *requireDirectoryExists*, which aligns with the refactoring operation conducted by the developers. However, such misidentifications are infrequent, and *references* are effective for matching code entities in most cases.

D. RQ2: Performance in Entity Matching

To measure the performance of the proposed algorithm 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. #MST=#FP+#FN (i.e., the number of false positives plus the number of false negatives) represents how frequently the evaluated matching algorithms make mistakes. The key concern of RQ2 is to what extent the proposed matching algorithm can reduce the frequency of mistakes (i.e., #MST). Besides #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.

Our evaluation results are presented in Fig. 12. The horizontal axis presents the involved projects. The vertical axis presents the number of false positives and false negatives as well as their sum (i.e., #FP, #FN, and #MST) on each subject project. From this figure, we make the following observations:

- The proposed matching algorithms substantially reduced the frequency of mistakes: The total number of mistakes (i.e., #MST) was reduced from 442 to 146, with a substantial reduction of 67%=(442-146)/442.
- On average, the number of false positives per project was reduced by 39.5%=(4.05-2.45)/4.9 and the number of false negatives per project was reduced by 74.8%=(18.05-4.55)/18.05.
- The proposed matching algorithms reduced or maintained the frequency of mistakes on most (17 out of the 20) of the subject projects.

We employed the Wilcoxon signed-rank test [59] and used Cliff's Delta (d) as the effect size [60] to validate whether there is a statistically significant difference between the total number of mistakes caused by the two approaches. The evaluation results (p-value=9.69E-5 and Cliff's $|d|$=0.42) confirmed that the reduction in the number of mistakes (#MST) was statistically significant. Similarly, the significance tests confirmed that the improvement in precision (p-value=0.04 and Cliff's $|d|$=0.19) and recall (p-value=3.09E-5 and Cliff's $|d|$=0.49) was statistically significant.

TABLE V
PERFORMANCE PER ENTITY TYPE

Entity Type	Approach	#MST	#FP	#FN	Precision	Recall
Classes	Our Algorithm	2	0	2	100%	99.62%
	Baseline	23	3	20	99.4%	96.16%
	Δ *Improvement*	*-21*	*-3*	*-18*	*0.6%*	*3.46%*
Interfaces	Our Algorithm	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ *Improvement*	*0*	*0*	*0*	*0%*	*0%*
Enums	Our Algorithm	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ *Improvement*	*0*	*0*	*0*	*0%*	*0%*
Annotation Types	Our Algorithm	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ *Improvement*	*0*	*0*	*0*	*0%*	*0%*
Initializers	Our Algorithm	0	0	0	100%	100%
	Baseline	2	0	2	100%	77.78%
	Δ *Improvement*	*-2*	*0*	*-2*	*0%*	*22.22%*
Fields	Our Algorithm	3	2	1	98.23%	99.11%
	Baseline	17	3	14	97.03%	87.5%
	Δ *Improvement*	*-14*	*-1*	*-13*	*1.2%*	*11.61%*
Methods	Our Algorithm	16	4	12	99.6%	98.8%
	Baseline	41	7	34	99.28%	96.59%
	Δ *Improvement*	*-25*	*-3*	*-22*	*0.32%*	*2.21%*
Enum Constants	Our Algorithm	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ *Improvement*	*0*	*0*	*0*	*0%*	*0%*
Annotation Members	Our Algorithm	0	0	0	100%	100%
	Baseline	0	0	0	100%	100%
	Δ *Improvement*	*0*	*0*	*0*	*0%*	*0%*
Statements	Our Algorithm	125	43	82	96.85%	94.16%
	Baseline	359	68	291	94.24%	79.27%
	Δ *Improvement*	*-234*	*-25*	*-209*	*2.61%*	*14.89%*
Total	Our Algorithm	146	49	97	98.39%	96.86%
	Baseline	442	81	361	97.12%	88.32%
	Δ *Improvement*	*-296*	*-32*	*-264*	*1.27%*	*8.54%*

We further investigated their performance in matching different categories of code entities, e.g., "classes", "methods" and "statements". The evaluation results are presented in Table V. From this table, we observe that the proposed matching algorithms outperform or are comparable to the baseline approach across all involved entity types. We also noticed that the proposed matching algorithms resulted in high precision and recall on all of the entity types. The minimal precision (on "statements") was 96.85%, and the minimal recall (on "statements") was 94.24%. It may suggest that the proposed matching algorithms worked well on all entity types. The evaluated approaches reported the highest numbers of mistakes in matching entities of "statements". It is reasonable because the number of involved statements is significantly greater than those of classes, methods, and fields.

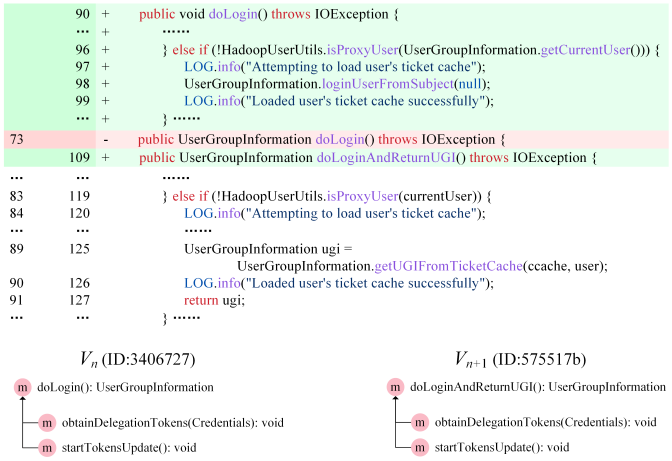

Fig. 13. False Positive Avoided by the Proposed Matching Algorithm

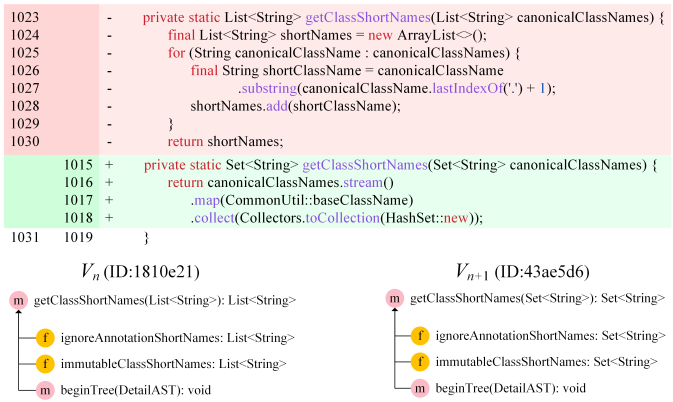

Fig. 14. True Positive Missed by the Selected Baseline

We leverage a real-world example from project Flink [65] in Fig. 13 to illustrate how the proposed matching algorithms avoided some false positives. The method *doLogin* on Line 73 in V_n was renamed as *doLoginAndReturnUGI* on Line 109 in V_{n+1}, and a new method named *doLogin* on Line 90 in V_{n+1} was added. However, the baseline approach mapped method *doLogin* in V_n by mistake to method *doLogin* in V_{n+1} because their method signatures were identical except for the return types. In contrast, our approach correctly mapped method *doLogin* in V_n to method *doLoginAndReturnUGI* in V_{n+1} because 1) their method bodies were exactly the same and 2) they were called by the same software entities, i.e., *startTokensUpdate* and *obtainDelegeationTokens*. It did not map *doLogin* in V_n to *doLogin* in V_{n+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 Checkstyle [66] in Fig. 14 to illustrate how the proposed matching algorithms successfully retrieved the pairs (i.e., false negatives) missed by the baseline approach. The baseline approach failed to match method *getClassShortNames* on Line 1023 in V_n with method *getClassShortNames* on Line 1015 in V_{n+1}. The change between these two methods involves a complex refactoring (*replace loop with pipeline* [1], [67]).

TABLE VI
IMPACT OF PATCH SIZE

Patch Size	REEXTRACTOR+			REFACTORINGMINER		
	#TP	#FP	#FN	#TP	#FP	#FN
Q1 [2, 30, 61]	1.88	0.22	0.36	1.36	0.46	0.88
Q2 [62, 102, 149]	2.97	0.21	0.6	2.38	0.25	1.19
Q3 [150, 268, 416]	6.8	0.5	0.92	5.37	1	2.35
Q4 [419, 1851, 32372]	18.49	0.81	2.12	16.34	2.37	4.27

The refactoring made the two method bodies dissimilar, which caused the failure of the baseline approach that heavily depended 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 (i.e., fields *ignoreAnnotationShortNamses* and *immutableClassShortNames* as well as method *beginTree*).

E. RQ3: 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 VI. 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 VI specifies the data groups along with the minimum size, average size, and maximum size of the patches within each data group. For example, within the first category Q1, the minimum size, average size, and maximum size of the patches are 2, 30, and 61, respectively. Columns 2 to 5 present the performance of REEXTRACTOR+ on the given data group. Columns 6 to 9 present the performance of the baseline approach on the given data group. Notably, the performance metrics are the averages. For example, #TP is the average number of true positives per commit in the given group.

We observe from Table VI that the evaluated approaches made more false positives and false negatives 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 VI, we also observe that REEXTRACTOR+ outperformed the baseline approach on all categories of the patches, i.e., Q1, Q2, Q3, and Q4. It may suggest that REEXTRACTOR+ can stably improve the state of the art in refactoring detection regardless of the patch size in the commits. However, as shown in Table VI, the patch size has a relatively greater impact on the performance of the baseline approach than that on REEXTRACTOR+. For REEXTRACTOR+, the average number of false positives varies slightly from 0.22 (Q1) to 0.81 (Q4) and the average number of false negatives varies somewhat from 0.36 (Q1) to 2.12 (Q4), suggesting that the patch size had little influence on the performance of REEXTRACTOR+.

To quantitatively measure the correlation between the patch size and the performance of the evaluated approaches, we computed their point-biserial correlation coefficients [68] by taking each commit (patch) as an individual. The computation

results suggested that the correlation coefficient between the patch size and our approach's false positives and false negatives were 0.14 and 0.12, respectively. It may indicate a weak correlation between the patch size and both false positives and false negatives in our approach. The correlation coefficients between the patch size and the baseline approach's false positives and false negatives were 0.5 and 0.32, respectively. It may suggest a moderate correlation between the patch size and false positives in the baseline approach, while the correlation between the patch size and its false negatives is weak.

F. RQ4: Efficiency

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 17.0.5 x64 with a maximum of 8GB Java heap memory (i.e., -Xmx8g). For each approach, we recorded the execution time taken to detect refactorings on a given repository and commit using `System.nanoTime()` Java method [69]. All repositories were first cloned locally from GitHub. To make a fair comparison, we followed the methodology outlined by Tsantalis et al. [34] to exclude the time spent on checkout commits, as the checkout operation involves disk writes that introduce significant time overhead. Specifically, it takes approximately 10.2 minutes to process all 400 commits by checking out commits, with a maximum checkout time of 8.62 seconds. Our evaluation results suggest that RefactoringMiner keeps efficient across all commits regardless of their patch size. The median and average execution time for all commits were 0.17 seconds and 0.63 seconds, respectively, with a minimum execution time of 0.01 seconds and a maximum execution time of 34.48 seconds. In comparison, our approach had median and average execution times of 0.88 seconds and 2.27 seconds, respectively, with a minimum execution time of 0.07 seconds and a maximum execution time of 37.89 seconds. The time cost of our approach is acceptable because large-scale refactoring discovery is usually conducted once to construct extensive datasets of real-world refactorings. As for the refactoring discovery on a specific commit or whole software project (to understand the history of code evolution), the proposed approach could finish in a few seconds (for a single commit) or tens of minutes (for a single project). Notably, the proposed entity matching algorithm should to checkout parent commits and retrieve the references of code entities in the two versions, as we will discuss later. It is a time-consuming process that consumes 65.5%=596.12/909.65 (seconds) of the total time cost.

We investigated the number of iterations required for the proposed matching algorithm to become stable. The evaluation results are presented in Table VII. The first column specifies the number of iterations performed. Column 2 presents how many (and how percentage) the iterative matching became stable after the given number of iterations. For example, according to the second row of the table, we know that the entity matching algorithm became stable on 260 commits (accounting for 65% of the evaluated commits) after two iterations. From the last column, we know that on 34 samples (method pairs), the statement matching algorithm required 5+ iterations to become stable. From Table VII, we observe that the iterative strategy in the proposed matching algorithms is efficient. In 90.5%=(65%+25.5%) of cases, the entity matching became stable after two or three iterations. The median and average number of iterations needed for entity mappings are 2 and 2.5, respectively. We also observe that the statement mappings had a great chance (90.69%) to become stable after only two iterations. The median and average number of iterations needed for statement mappings are 2 and 2.13, respectively. The maximum number of iterations needed for both entity mappings and statement mappings is eight.

TABLE VII
Executed Iterations for Iterative Matching

Required Iterations	Entity Matching	Statement Matching
2	260 (65%)	6621 (90.69%)
3	102 (25.5%)	538 (7.37%)
4	25 (6.25%)	85 (1.16%)
5	6 (1.5%)	23 (0.32%)
5+	7 (1.75%)	34 (0.47%)

IV. Discussion

To compute the reference-based similarity, we build an ERG to retrieve the references associated with code entities. Notably, we exclude the AST nodes from third-party libraries, as such nodes do not serve as reference relations of the to-be-matched entities. In other words, the code in third-party libraries does not call any methods or create objects from the local project, thus there are no references to the local code entities within them. In addition, our approach leverages the binding information of AST nodes to associate with code entities. Notably, the proposed approach needs to checkout both the parent and child commits to retrieve reference relations of the code entities in the old and the new versions respectively. This step is necessary because the `resolveBinding()` method depends on the project's environment, including class path, source path, and encoding. If the project's version on the hard disk differs from the version corresponding to the to-be-retrieved entities, JDT cannot resolve the source path, and thus the method will return `null`. As a result, our approach incurs a higher time cost compared to the baseline approach.

We propose an efficient retrieval strategy to build the ERG, i.e., reference relations are not retrieved from unchanged source code files. First, for some large projects, the number of unchanged files may be thousands or even more. It is resource-intensive and time-consuming to retrieve reference relations from a large number of source code files. Second, although this strategy may miss some references, it does not impact any performance of the matching algorithm. We imagine two scenarios to explain why the proposed retrieval strategy does not impact the performance:

1) For code entities (e.g., methods) whose signatures and enclosing classes remain unchanged, their references may exist in the unchanged source code files. However, such entities have already been matched during the qualified name-based matching and will not be matched again during the reference-based matching. Therefore,

there is no need to retrieve references from unchanged files for such entities.

2) For code entities whose signatures or enclosing classes have changed, their references cannot exist in unchanged source code files. For example, if a method's signature changes (e.g., renaming or adding parameters), the entities that directly call it must also change, and the files in which they are located will also change.

In summary, the proposed approach does not need to retrieve references from unchanged files, improving efficiency without reducing performance.

Our approach demonstrates superior performance in identifying class-level refactorings, such as *rename class* and *move class*, significantly outperforming existing approaches. The success is primarily due to our approach's capability to leverage the references of the entities to match code changes and infer these refactorings. However, we also observed that REFACTORINGMINER may erroneously identify *move method* refactorings within renamed or moved classes because it fails to match them. A particularly challenging scenario arises when the to-be-matched classes are empty (i.e., classes that do not contain any code entities) and lack external references. Our approach first utilizes the JGit RenameDetector, as introduced in Section II-B, to establish an initial mapping for such classes. After that, the various similarities (including reference-based similarity) are employed to update the mapping. For empty classes that lack external references, the mapping built by JGit RenameDetector is the final mapping and no more adjustment is applied. In addition, future work will attempt to incorporate additional references such as *StringLiteral* to further overcome the limitation of missing external references.

Moreover, our approach tends to accurately identify cases where a small expression is extracted from a larger statement into a new method. We leverage a real-world example from project Commons IO [70], as depicted in Fig. 15, to illustrate why our approach could identify such refactorings. In this example, the developer extracted the method call *getClass().getSimpleName()* on Line 603 in V_n as a new method *getSimpleName* on Line 577 in V_{n+1}. Our approach could successfully identify this refactoring since the refactored statement becomes textually identical to the original statement after replacing the method call with the return expression.

For identifying overlapping refactorings, our approach could effectively handle such cases by employing a novel heuristic that computes the similarity between the extracted expression and the original statement, i.e., $|M| > |U_{T_{v,i}}|$. For example, the developer extracted a method call expression into a new variable and renamed the called method. Although the developer conducted overlapping refactorings, our approach could still identify them (i.e., *extract variable* and *rename method* refactorings) based on the heuristic rules defined in Section II-G. The evaluation results demonstrate the effectiveness of our approach in identifying such refactorings. However, it is important to note that the current state of our approach does not yet support nested refactorings, such as *extract method* within another extracted method. This limitation highlights an area for future improvement and refinement.

```
        577 +     public String getSimpleName() {
        578 +         return getClass().getSimpleName();
        579 +     }
601     605       @Override
602     606       public String toString() {
603         -         return getClass().getSimpleName() + "[" + origin.toString() + "]";
        607 +         return getSimpleName() + "[" + origin.toString() + "]";
604     608       }
```

V_n (ID:3c53c39) V_{n+1} (ID:7ea97b3)

Fig. 15. True Positive Identified by Our Approach

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 refactoring experts to manually check the refactorings involved in the selected commits (from open-source projects). However, the participants were not the original developers, and thus their manual checking could be inaccurate. To mitigate this threat, we requested them to independently check each refactoring, and computed the Fleiss' kappa coefficient (0.82) to validate the high level of consistency. If the evaluated tools detect the same changes but report them with different types of refactoring, the evaluation may be biased, especially for refactorings such as *Split Method*, *Merge Method*, *Split Class*, and *Merge Class*. To mitigate this threat, we applied RefactoringMiner to identify all instances of these four refactoring types and carefully validated them to prevent RefactoringMiner from being penalized with false negatives.

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 datasets may suffer from limited generality. To increase the diversity of the dataset, we selected 20 refactoring-discovering commits from each of the 20 subject projects. On one side, the testing data covers diverse projects, increasing the diversity of the dataset. On the other side, it limited the total number of to-be-checked refactorings, which reduced the cost of manual checking.

Another threat to the external validity of our conclusions lies in the sampling method for selecting the 400 commits analyzed in this paper. While these commits were not chosen randomly, the selection process was carefully designed to ensure the experiment's reproducibility. However, this controlled selection may introduce bias and limit the generalizability of our findings to the broader population of commits. Future work could explore more diverse and randomized datasets to mitigate this threat. In addition, `RefactoringMiner` supports over 100 types of refactoring, but our evaluation only compares the results on 28 of these refactoring types. These refactorings were chosen because they are among the most popular, widely used by developers, and extensively studied in the research community. However, the limited scope may overlook potential discrepancies or insights associated with other refactoring types. To mitigate this threat, in the future, we would like to implement more refactoring heuristics to

support a broader range of refactoring types and conduct a more comprehensive comparison with competitive tools.

To facilitate further validation and further improvement, we make the replication package (including both the implementation of the proposed approach and the evaluation data) publicly available on GitHub [71].

B. Limitations

Although the key insight of the proposed approach is not language-specific, the current implementation of the approach works on Java projects only. First, it depends on automated parsing of source code that is often language-specific. Second, the proposed approach leverages the data types of code entities, and thus it may not be easily adapted to programming languages like *Python* and *JavaScript* where the data types could not be determined via static analysis. Finally, some heuristics used for refactoring detection are also language-specific, such as *pull up* and *push down* refactorings.

The reference-based approach is more resource-consuming than alternatives because it is time-consuming to retrieve all references for each of the involved entities within a given commit. To minimize the cost, we take the following measures. First, our approach 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, the proposed approach 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. Ignoring such unchanged files could significantly reduce the cost because most of the files in the repository are not influenced by a given commit.

References play a crucial role in matching code entities, but their effectiveness may be limited or even backfire, particularly in cases involving unreliable references. As presented in Fig 11, unreliable references can lead to incorrect matches especially when the implementations of code entities exhibit significant structural differences. Furthermore, in cases where code entities lack references altogether, we rely solely on their implementations for matching, highlighting the limitations of reference-based matching. We observed that in these specific cases, the entity matching algorithm based on AST node replacements (i.e., `RefactoringMiner` [33], [34]) tends to match more accurately. This is likely because AST node replacement-based matching focuses on structural similarity between code entities, independent of external references, which can sometimes be missing. A typical example from project Commons Lang [72] is presented in Fig. 16. In this example, our approach was unable to match test method *testTimeZoneStrategy* on Line 57 in V_n with test method *testTimeZoneStrategy_TimeZone* on Line 63 in V_{n+1}. The mismatching lies in the absence of references and the low implementation similarity between the two methods. However, REFACTORINGMINER successfully identifies a matching between the statements on Line 58 and Line 64 within the respective methods by replacing the method call node. As a result, these two methods can be matched because their statements become textually identical after the replacement.

```
55  61      @ParameterizedTest
56  62      @MethodSource("java.util.Locale#getAvailableLocales") {
57      -   public void testTimeZoneStrategy(final Locale locale) {
58      -       testTimeZoneStrategyPattern(locale);
        63  +   public void testTimeZoneStrategy_TimeZone(final Locale locale) {
        64  +       testTimeZoneStrategyPattern_TimeZone_getAvailableIDs(locale);
59  65      }
66      -   public void testTimeZoneStrategyPattern(final Locale locale) {
        78  +   public void testTimeZoneStrategyPattern_TimeZone_getAvailableIDs(
                                                        final Locale locale) {
67  79          Objects.requireNonNull(locale, "locale");
... ...         ......
59  65      }
```

V_n (ID:56e6687) V_{n+1} (ID:880b85d)

Fig. 16. False Negative Due to the Absence of References

Another limitation of our approach is that it has a slower execution time compared to the baseline approach. However, our evaluation results demonstrate that the performance improvement achieved by our approach compensates for this drawback. Consequently, for developers who prioritize speed and need immediate results, the baseline approach may be a more suitable choice. Nevertheless, it is important to note that refactoring discovery is performed in the backend, where execution time is usually less critical.

V. RELATED WORK

A. Refactoring Detection

Identifying software refactorings is crucial for understanding the evolution of software systems and facilitating research in the field with real-world instances. Several approaches have been proposed for refactoring detection. These approaches typically involve matching code entities between two versions of a software project and applying heuristics to identify refactorings. One such approach is REFACTORINGCRAWLER developed by Dig et al. [31], which first employs *Shingles encoding* [73] for syntactic analysis to discover similar code entities (methods, classes, and packages) and identify potential refactoring from these pairs of similar entities rapidly. It then uses semantic analysis to verify these refactorings, making it a reliable approach. The proposed approach differs from it in that our approach leverages fully qualified names to match intact (without any changes) entities and entities with the same identity (including qualified name, method signatures, and data types. UMLDIFF [30] is a structural differencing algorithm designed to automatically detect elementary structural changes in software components (such as packages, classes, and methods) by comparing the design models of different system versions. Xing and Stroulia [74] proposed JDEVAN [75], which detects and categorizes refactorings based on design-level changes reported by UMLDIFF. Silva and Valente [11], [32] proposed REFDIFF that utilizes static analysis and code similarity to detect various refactorings. It begins by tokenizing the source code of the project. Each code element (such as classes, methods, and fields) is transformed into a bag of

tokens. For each token in the codebase, it computes the weight using a variation of the TF-IDF weighting scheme [76], which assigns higher importance to tokens that are less frequent in the codebase, as these tokens are often more discriminative. To match code elements between the two revisions, REFDIFF employs a weighted Jaccard coefficient [77] to compute the similarity between code elements, and detects refactorings based on their similarity and defined thresholds. Note that such approaches are limited to matching coarse-grained code entities (e.g., classes and methods) and do not support the detection of low-level within-method refactorings (e.g., *extract variable* refactorings).

To identify refactorings, REF-FINDER proposed by Prete et al. [78], [79] encodes code elements (e.g., classes, methods, and fields) and their relationships using logic predicates. These predicates describe code elements and their containment relationships within the codebase. In addition, each refactoring type is encoded as a logic rule, where the antecedent predicates represent prerequisites or change facts, specifying the conditions that must be met for inferring a refactoring, and the consequent predicate represents the target refactoring type to be detected. REF-FINDER infers refactoring instances by converting the antecedent of each rule into a logic query and applying this query to the database of logic facts. When the query conditions are satisfied, the approach identifies a refactoring instance based on the rule's consequent predicate. Note that REF-FINDER supports the detection of several low-level refactorings, e.g., *inline variable* refactorings. REFACTORINGMINER proposed by Tsantalis et al. [33], [34] is the only approach that supports both high-level and low-level refactorings and operates at commit level. At the heart of REFACTORINGMINER is a statement matching algorithm without relying on any similarity thresholds, which is used to identify and match code changes involved in refactorings. Based on the matched entities, REFACTORINGMINER presents a set of refactoring detection rules to map code changes and high-level refactorings. In addition, it depends on AST node replacements in statement mappings to infer low-level refactorings within methods. The proposed approach differs from the existing approaches introduced in the preceding paragraphs in that our approach takes full advantage of the less-exploited references of code entities and the contexts of the to-be-matched statements.

B. Entity Matching

Entity matching originated from "origin analysis" pioneered by Godfrey et al. [80]–[82]. Existing entity matching algorithms could be categorized into two main groups. The first category focuses on mapping one design model into another, taking advantage of two design models as input without delving into the detailed implementation of code entities, such as method bodies. A prominent example of this category is UMLDIFF, proposed by Xing and Stroulia [30]. Although it takes the source code as input, it actually works on the class diagrams automatically generated from the source code. Consequently, it accomplishes the mapping of code entities by correlating one graph with another, where a graph represents a class diagram. In this context, nodes within the graph represent code entities, and edges depict the relations between these entities. The matching process is based on their labels' similarity (i.e., name-based similarity) and similarity in connections (i.e., edges). Another algorithm within this category is SIDIFF, proposed by Kelter et al. [83]. It initially matches leaf entities, like methods, based on their types, names, and signatures (e.g., parameters of methods), and then adopts a bottom-up tactic to match high-level entities, like classes, by evaluating their similarity in names, matched sub-entities, shared generalization targets, and common packages. Once a high-level entity is matched, it employs a top-down tactic to match its sub-entities. Kim et al. [84] presented an approach to automatically infer changes at or above method headers by defining a set of low-level transformations (e.g., altering return types of methods) on method headers. Two methods are considered to be matched if a sequence of rules can transform one method's header into that of the other.

The second category fully exploits the source code involved in the matching process, yet none of these matching algorithms exploit the references of code entities, as our approach does. JDIFF, proposed by Apiwattanapong et al. [85], [86], falls under this category. It begins by matching classes and interfaces based on their fully qualified names and then matches methods within these pairs of classes/interfaces according to method signatures. To identify the difference between matched methods, it represents the method bodies as enhanced control flow graphs and matches nodes in these graphs through graph isomorphism. CHANGEDISTILLER, proposed by Fluri et al. [40], is widely utilized for fine-grained extraction of source code changes. Given two Abstract Syntax Trees (ASTs) representing the source code before and after the revision, CHANGEDISTILLER identifies a sequence of changes that can transform one tree into the other. The core of the algorithm lies in the similarity-based matching between two AST nodes, where the similarity between nodes indicates the percentages of common descendant nodes. GUMTREE, proposed by Falleri et al. [41], improves upon CHANGEDISTILLER by eliminating the assumption that leaf nodes contain significant text. It employs a greedy top-down search algorithm to discover the greatest isomorphic subtrees between two ASTs and employs a bottom-up algorithm to match code entities based on common (matched) subtrees. REFDIFF, proposed by Silva et al. [11], [32], leverages a variation of the TF-IDF weighting scheme [76] and a weighted Jaccard coefficient [77] to compute the similarity between code entities (considering them as plain text), and match entities based on predefined similarity thresholds. IASTMAPPER [87], proposed by Zhang et al., is an iterative similarity-based AST mapping algorithm to locate the code changes. Notably, although they also exploit the contexts of statements to facilitate statement matching, the contexts of the to-be-matched statements contain only the nearest above and the nearest below sibling statements. In contrast, our approach exploits all other statements in the innermost enclosing block of the to-be-matched statements as contexts. REFACTORINGMINER, proposed by Tsantalis et al. [33], [34], matches code entities without applying any similarity thresholds, greatly simplifying the usage of the

algorithm. It employs a top-down tactic, starting from classes and proceeding to methods and fields, to match code entities with identical signatures. For the remaining code entities that involve signature changes or refactorings, it adopts a bottom-up tactic, starting from methods and proceeding to classes, to match them. At its core, it employs a novel statement matching algorithm to match two statements by measuring the AST node-based edit distance [43] between them. Alikhanifard and Tsantalis improved the accuracy of statement mappings in RefactoringMiner 3.0 [88]. For leaf statements (statements without a body), their approach employs three successive rounds of candidate sorting based on string edit distance, node depth difference, and positional index difference in their parent's children list. For composite statements (those with nested statements), the same criteria are applied with an additional metric: the ratio of matched children. To further refine the matching process, they developed a novel sorting function that selects the best candidate when multiple potential matches satisfy these criteria. The proposed context-aware algorithm differs from the existing statement matching algorithms, like iASTMapper [87] and RefactoringMiner [33], [34], [88], in that it takes full advantage of the other statements within the innermost enclosing block as context-based similarity to facilitate statement mappings.

VI. Conclusion

In this paper, we presented a novel refactoring detection approach (called ReExtractor+) that supports the detection of both high-level and low-level refactorings. It leverages a reference-based entity matching algorithm to match coarse-grained code entities between two successive versions. The entity matching algorithm takes full advantage of qualified names, implementations, and references. ReExtractor+ also leverages a context-aware statement matching algorithm to match statements between the pairs of matched methods/initializers. The statement matching algorithm exploits the contexts of statements to facilitate statement matching. Our evaluation results on real-world software applications suggested that ReExtractor+ improves the state of the art in refactoring detection: It reduces the number of false positives by 57.4% and improves recall by 18.4%.

The implementation of the proposed approach is currently confined to Java. In the future, we would like to extend it to other 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 identify additional types of refactorings with ReExtractor+.

Acknowledgment

The authors would like to thank the anonymous reviewers from IEEE TSE and ASE for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037), China Postdoctoral Science Foundation (No. 2023M740078), and China National Postdoctoral Program for Innovative Talents (BX20240008).

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[2] X. Chen, X. Hu, Y. Huang, H. Jiang, W. Ji, Y. Jiang, Y. Jiang, B. Liu, H. Liu, X. Li *et al.*, "Deep learning-based software engineering: progress, challenges, and opportunities," *Science China Information Sciences*, vol. 68, no. 1, pp. 1–88, 2025, https://doi.org/10.1007/s11432-023-4127-5.

[3] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013, https://doi.org/10.1109/TSE.2013.4.

[4] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2018, https://doi.org/10.1109/TSE.2017.2658573.

[5] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. Buenos Aires, Argentina: IEEE, 2017, pp. 176–185, https://doi.org/10.1109/ICPC.2017.38.

[6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16')*, 2016, pp. 858–870, https://doi.org/10.1145/2950290.2950305.

[7] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009, https://doi.org/10.1109/TSE.2009.1.

[8] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012, https://doi.org/10.1016/j.jss.2012.04.013.

[9] H. Liu, Q. Liu, Y. Liu, and Z. Wang, "Identifying renaming opportunities by expanding conducted rename refactorings," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 887–900, 2015, https://doi.org/10.1109/TSE.2015.2427831.

[10] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: A large-scale survey of software refactoring," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*, 2021, pp. 1303–1313, https://doi.org/10.1145/3468264.3473924.

[11] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020, https://doi.org/10.1109/TSE.2020.2968072.

[12] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Honolulu, HI, USA: ACM, 2011, pp. 151–160, https://doi.org/10.1145/1985793.1985815.

[13] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2012, https://doi.org/10.1109/TSE.2012.63.

[14] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, "On the impact of refactoring on the relationship between quality attributes and design metrics," in *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*. IEEE, 2019, pp. 1–11, https://doi.org/10.1109/ESEM.2019.8870177.

[15] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Software Engineering*, vol. 25, pp. 4315–4347, 2020, https://doi.org/10.1007/s10664-020-09871-2.

[16] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, and M. Aniche, "Data-driven extract method recommendations: A study at ING," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE '21)*, 2021, pp. 1337–1347, https://doi.org/10.1145/3468264.3473927.

[17] A. Ketkar, O. Smirnov, N. Tsantalis, D. Dig, and T. Bryksin, "Inferring and applying type changes," in *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Pittsburgh, PA, USA: ACM, 2022, pp. 1206–1218, https://doi.org/10.1145/3510003.3510115.

[18] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC '19)*. IEEE, 2019, pp. 53–64, https://doi.org/10.1109/ICPC.2019.00019.

[19] B. Liu, H. Liu, G. Li, N. Niu, Z. Xu, Y. Wang, Y. Xia, Y. Zhang, and Y. Jiang, "Deep learning based feature envy detection boosted by real-world examples," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23*. IEEE, 2023, pp. 1–13, https://doi.org/10.1145/3611643.3616353.

[20] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. IEEE, 2012, pp. 411–416, https://doi.org/10.1109/CSMR.2012.79.

[21] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2014, https://doi.org/10.1109/TSE.2014.2372760.

[22] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. De Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*, 2017, pp. 465–475, https://doi.org/10.1145/3106237.3106259.

[23] E. L. Alves, M. Song, T. Massoni, P. D. Machado, and M. Kim, "Refactoring inspection support for manual refactoring edits," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 365–383, 2017, https://doi.org/10.1109/TSE.2017.2679742.

[24] E. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages," in *Proceedings of the 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR '19)*. IEEE, 2019, pp. 51–58, https://doi.org/10.1109/IWoR.2019.00017.

[25] W. Wang and M. W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14*. IEEE, 2014, pp. 331–340, https://doi.org/10.1109/ICSME.2014.55.

[26] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, "The effectiveness of supervised machine learning algorithms in predicting software refactoring," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1432–1450, 2020, https://doi.org/10.1109/TSE.2020.3021736.

[27] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*. IEEE, 2006, pp. 231–240, https://doi.org/10.1109/ASE.2006.41.

[28] H. Liu, Z. Niu, Z. Ma, and W. Shao, "Identification of generalization refactoring opportunities," *Automated Software Engineering*, vol. 20, pp. 81–110, 2013, https://doi.org/10.1007/s10515-012-0100-0.

[29] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, 2012, pp. 1–11, https://doi.org/10.1145/2393596.2393655.

[30] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. Long Beach, CA, USA: ACM, 2005, pp. 54–65, https://doi.org/10.1145/1101908.1101919.

[31] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*. Nantes, France: Springer, 2006, pp. 404–428, https://doi.org/10.1007/11785477_24.

[32] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. Buenos Aires, Argentina: IEEE, 2017, pp. 269–279, https://doi.org/10.1109/MSR.2017.14.

[33] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden: ACM, 2018, pp. 483–494, https://doi.org/10.1145/3180155.3180206.

[34] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022, https://doi.org/10.1109/TSE.2020.3007722.

[35] B. Liu, H. Liu, N. Niu, Y. Zhang, G. Li, and Y. Jiang, "Automated software entity matching between successive versions," in *Proceedings of the 38th IEEE International Conference on Automated Software Engineering (ASE '23)*. Kirchberg, Luxembourg: IEEE, 2023, pp. 1615–1627, https://doi.org/10.1109/ASE56229.2023.00132.

[36] "Eclipse JDT ASTNode.toString()," https://help.eclipse.org/latest/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTNode.html&anchor=toString(), 2024.

[37] "Eclipse JGit RenameDetector," https://archive.eclipse.org/jgit/docs/jgit-2.3.1.201302201838-r/apidocs/org/eclipse/jgit/diff/RenameDetector.html, 2024.

[38] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945, https://doi.org/10.2307/1932409.

[39] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the 1988 International Conference on Software Maintenance (ICSM '98)*. Bethesda, MD, USA: IEEE, 1998, pp. 368–377, https://doi.org/10.1109/ICSM.1998.738528.

[40] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change Distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007, https://doi.org/10.1109/TSE.2007.70731.

[41] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Vasteras, Sweden: ACM, 2014, pp. 313–324, https://doi.org/10.1145/2642937.2642982.

[42] G. W. Adamson and J. Boreham, "The use of an association measure based on character structure to identify semantically related pairs of words and document titles," *Information Storage and Retrieval*, vol. 10, no. 7-8, pp. 253–260, 1974, https://doi.org/10.1016/0020-0271(74)90020-5.

[43] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[44] "Expression statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FExpressionStatement.html, 2024.

[45] "Return statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FReturnStatement.html, 2024.

[46] "Variable declaration statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FVariableDeclarationStatement.html, 2024.

[47] "Assignment," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAssignment.html, 2024.

[48] "For statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FForStatement.html, 2024.

[49] "Enhanced for statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FEnhancedForStatement.html, 2024.

[50] "While statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FWhileStatement.html, 2024.

[51] "Do statement," https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FDoStatement.html, 2024.

[52] N. Tsantalis, "The implementation of RefactoringMiner (release: 3.0.7)," https://github.com/tsantalis/RefactoringMiner, 2024.

[53] "Refactoring oracle," https://github.com/tsantalis/RefactoringMiner/blob/master/src/test/resources/oracle/data.json, 2024.

[54] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "CodeShovel: Constructing method-level source code histories," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*. Madrid, Spain: IEEE, 2021, pp. 1510–1522, https://doi.org/10.1109/ICSE43902.2021.00135.

[55] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychological Bulletin*, vol. 76, no. 5, pp. 378–382, 1971, https://doi.org/10.1037/h0031619.

[56] T. Junk, "Confidence level computation for combining searches with small statistics," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 434, no. 2, pp. 435–443, 1999, https://doi.org/10.1016/S0168-9002(99)00498-2.

[57] "Netty," https://github.com/netty/netty/commit/303cb535239a6f07cbe24a033ef965e2f55758eb#diff-0b07dfa3c28036ef062f7860b9e87a971bb82cac9f427fceac817d0fdc02af50L1461, 2024.

[58] "Zuul," https://github.com/Netflix/zuul/commit/b25d3f32ed2e2da86f5c746098686445c2e2a314#diff-a2e9124ad6a21e1bcbaeee340126b0e5c698ac1f8820dc271491d0ff571bfc8bL193, 2024.

[59] F. Wilcoxon, "Individual comparisons by ranking methods," *International Biometric Society*, vol. 1, no. 6, pp. 80–83, 1945, https://doi.org/10.2307/3001968.

[60] R. J. Grissom and J. J. Kim, *Effect Sizes for Research: A Broad Practical Approach*. Hillsdale, NJ: Lawrence Erlbaum Associates, 2005.

[61] S. McKillup, *Statistics Explained: An Introductory Guide for Life Scientists*. Cambridge, UK: Cambridge University Press, 2006.

[62] "JavaParser," https://github.com/javaparser/javaparser/commit/e405cee, 2024.

[63] "Checkstyle," https://github.com/checkstyle/checkstyle/commit/a8f0657, 2024.

[64] "Commons IO," https://github.com/apache/commons-io/commit/33db63489777c9782f054f50b434c70a528527d7#diff-0bd8160a11e12ea1f96476a00e78d0a95b965d07c7fe126e1af498c08511dd70L2732, 2024.

[65] "Flink," https://github.com/apache/flink/commit/575517bbb8de36b21632e54b441b7dcbc4d061c4#diff-45c9720456d17ae7c5b5d825ef2d58a3e2d30f78bcadbceacb7a35a760288aeeL73, 2024.

[66] "Checkstyle," https://github.com/checkstyle/checkstyle/commit/43ae5d651d5b3078d9c04a0539134e811f461f5c#diff-eb159ca4c068124acd5ef9f9d88e86b478ebf0b2a079d8aec61b4c8ced305483L1023, 2024.

[67] M. Fowler, "The catalog of refactorings," https://refactoring.com/catalog/, 2024.

[68] R. F. Tate, "Correlation between a discrete and a continuous variable. point-biserial correlation," *The Annals of Mathematical Statistics*, vol. 25, no. 3, pp. 603–607, 1954, https://doi.org/10.1214/aoms/1177728730.

[69] "System.nanoTime() Java method," https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#nanoTime(), 2024.

[70] "Commons IO," https://github.com/apache/commons-io/commit/7ea97b34f0f75e88773947e71120e8e75165f781#diff-62674830cfda0821271a5ce932bf3c04fbc48ea11272cde7a8daf983f16b6884L603, 2024.

[71] "Replication package," https://github.com/bitselab/ReExtractorPlus, 2024.

[72] "Commons Lang," https://github.com/apache/commons-lang/commit/880b85da4a13cbf4cd74febea270f710419e75e5#diff-b60f2101669b77d480ac3d18298e322448088ee883567960c0d3a0279d58f7eeL57, 2024.

[73] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of SEQUENCES (SEQUENCES '97)*. IEEE, 1997, pp. 21–29, https://doi.org/10.1109/SEQUEN.1997.666900.

[74] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*. Benevento, Italy: IEEE, 2006, pp. 263–274, https://doi.org/10.1109/WCRE.2006.48.

[75] ——, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. Leipzig, Germany: ACM, 2008, pp. 951–952, https://doi.org/10.1145/1370175.1370203.

[76] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, 1986.

[77] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, "Finding the Jaccard median," in *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. Austin, TX, USA: SIAM, 2010, pp. 293–311, https://doi.org/10.1137/1.9781611973075.25.

[78] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. Timisoara, Romania: IEEE, 2010, pp. 1–10, https://doi.org/10.1109/ICSM.2010.5609577.

[79] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. Santa Fe, NM, USA: ACM, 2010, pp. 371–372, https://doi.org/10.1145/1882291.1882353.

[80] M. W. Godfrey and Q. Tu, "Tracking structural evolution using origin analysis," in *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*. Orlando, FL, USA: ACM, 2002, pp. 117–119, https://doi.org/10.1145/512035.512062.

[81] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*. Paris, France: IEEE, 2002, pp. 127–136, https://doi.org/10.1109/WPC.2002.1021334.

[82] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005, https://doi.org/10.1109/TSE.2005.28.

[83] U. Kelter, J. Wehren, and J. Niere, "A generic difference algorithm for UML models," *Software Engineering*, pp. 105–116, 2005.

[84] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. Minneapolis, MN, USA: IEEE, 2007, pp. 333–343, https://doi.org/10.1109/ICSE.2007.20.

[85] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. Linz, Austria: IEEE, 2004, pp. 2–13, https://doi.org/10.1109/ASE.2004.1342719.

[86] ——, "JDiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007, https://doi.org/10.1007/s10515-006-0002-0.

[87] N. Zhang, Q. Chen, Z. Zheng, and Y. Zou, "iASTMapper: An iterative similarity-based abstract syntax tree mapping algorithm," in *Proceedings of the 38th IEEE International Conference on Automated Software Engineering (ASE '23)*. Kirchberg, Luxembourg: IEEE, 2023, pp. 863–874, https://doi.org/10.1109/ASE56229.2023.00178.

[88] P. Alikhanifard and N. Tsantalis, "A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *ACM Transactions on Software Engineering and Methodology*, 2024, https://doi.org/10.1145/3696002.