RefactoringMiner 2.0

Nikolaos Tsantalis, *Senior Member, IEEE*, Ameya Ketkar, and Danny Dig, *Member, IEEE*

Abstract—Refactoring detection is crucial for a variety of applications and tasks: (i) empirical studies about code evolution, (ii) tools for library API migration, (iii) code reviews and change comprehension. However, recent research has questioned the accuracy of the state-of-the-art refactoring mining tools, which poses threats to the reliability of the detected refactorings. Moreover, the majority of refactoring mining tools depend on code similarity thresholds. Finding universal threshold values that can work well for all projects, regardless of their architectural style, application domain, and development practices is extremely challenging. Therefore, in a previous work [N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, Accurate and efficient refactoring detection in commit history, in 40th International Conference on Software Engineering, 2018, pp. 483–494], we introduced the first refactoring mining tool that does not require any code similarity thresholds to operate. In this work, we extend our tool to support low-level refactorings that take place within the body of methods. To evaluate our tool, we created one of the most accurate, complete, and representative refactoring oracles to date, including 7,226 true instances for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools, and validated by one up to four refactoring experts. Our evaluation showed that our approach achieves the highest average precision (99.6 percent) and recall (94 percent) among all competitive tools, and on median is 2.6 times faster than the second faster competitive tool.

Index Terms—Refactoring mining, refactoring oracle, precision, recall, execution time, git, commit

1 Introduction

Refactoring is a very common practice that helps developers to complete maintenance tasks (i.e., implement new features and fix bugs), and eliminate various design and code smells [2]. Many researchers empirically investigated the benefits of refactoring based on refactoring operations collected from open source projects by studying how the renaming of identifiers affects code readability [3], how and why developers rename identifiers [4], the impact of refactoring on code naturalness [5], the impact of refactoring on code smells [6], the co-occurrence of refactoring and self-admitted technical debt removal [7], and how the introduction of Lambda expressions affects program comprehension [8].

Despite the benefits, in some contexts, refactoring is perceived as *change noise*, which makes more difficult the completion of various software evolution related tasks. For instance, refactoring operations can cause merge conflicts when merging development branches [9], distract developers when reviewing behavior-altering changes [10], make bug-inducing analysis algorithms (e.g., SZZ [11], [12], [13]) to erroneously flag behavior-preserving changes (i.e., refactorings) as bug-introducing [14], cause breaking changes to clients of libraries and frameworks [15], cause unnecessary test executions for behavior-preserving changes [16]. For this reason, many *refactoring-aware* techniques have been developed to merge branches [17], [18], detect bug-introducing changes [19], adapt client software to library and framework updates [20], [21], [22], select regression tests [23], [24], and assist code review [25], [26], [27] in the presence of refactoring operations.

Both refactoring-aware tool builders and empirical researchers need accurate refactoring information at the finest level of code evolution granularity (i.e., commit level) to improve the efficacy of their tools and draw safer conclusions for their research questions, respectively. In our previous work [1], we developed a refactoring mining tool and showed that it has superior accuracy and faster execution time than a competitive tool, RefDiff [28].

RefactoringMiner 2.0 builds upon its predecessor (version 1.0) to support the detection of low-level or submethod-level refactorings (i.e., taking place within the body of a method), such as Rename/Extract Variable. According to Murphy-Hill *et al.* [29] submethod-level refactorings are not supported by the vast majority of refactoring detection tools, despite the fact that developers tend to apply such refactorings more frequently than high-level refactorings.

RefactoringMiner takes as input two revisions (i.e., a commit and its parent in the directed acyclic graph that models the commit history in git-based version control repositories) of a Java project, and returns a list of refactoring operations applied between these two revisions. It supports the detection of 40 refactoring types for 6 different kinds of code elements (i.e., packages, type declarations, methods, fields, local variables/parameters, and type references). The list of supported refactoring types is shown in Table 1 highlighting those present in Fowler's catalog [30], supported by IDEs, and applied manually by developers.

Unlike other existing refactoring detection approaches, such as Ref-Finder [31], RefactoringCrawler [32], and JDEvAn [33], which analyze all files in two versions of a Java

- *Nikolaos Tsantalis is with the Department of Computer Science and Software Engineering, Concordia University, Montreal H3G 1M8, QC, Canada. E-mail: nikolaos.tsantalis@concordia.ca.*
- *Ameya Ketkar is with the School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR USA. E-mail: ketkara@oregonstate.edu.*
- *Danny Dig is with the Department of Computer Science, University of Colorado, Boulder, CO 80309 USA. E-mail: digd@eecs.oregonstate.edu.*

Manuscript received 19 Dec. 2019; revised 22 May 2020; accepted 3 July 2020. Date of publication 8 July 2020; date of current version 15 Mar. 2022. (Corresponding author: Nikolaos Tsantalis.) Recommended for acceptance by M. Kim. Digital Object Identifier no. 10.1109/TSE.2020.3007722

TABLE 1
Refactoring types supported by RefactoringMiner versions

1.0 (15 refactoring types)	**2.0** (+25 refactoring types, 40 in total)
Extract Method	Extract Class
Inline Method	Extract Subclass
Rename Method	Extract Variable/Field
Move Method/Field	Rename Variable/Parameter/Field
Pull Up Method/Field	Move & Rename Method/Field
Push Down Method/Field	Move & Inline Method
Extract Superclass	Parameterize Variable
Extract Interface	Inline Variable
Move Class	Merge Variable/Parameter/Field
Rename Class	Split Variable/Parameter/Field
Extract & Move Method	Move & Rename Class
Change Package	Replace Variable/Field with Field
	Change Variable/Parameter/Return/Field Type

Present in Fowler's catalog [30] Supported in IDEs Manual change

project, RefactoringMiner analyzes only the added, deleted, and changed files between the two revisions. This makes RefactoringMiner not only more efficient, because it has less code elements to analyze and compare, but also more accurate, because the number of code element combinations to be compared is significantly less, thus reducing the probability of incorrect code element matches.

In addition, RefactoringMiner is the first refactoring detection tool that does not rely on code similarity thresholds. We utilize novel techniques, such as *abstraction* and *argumentization* to deal with changes taking place in code statements due to refactoring mechanics, and apply syntax-aware replacements of AST nodes when matching two statements to deal with overlapping refactorings (e.g., variable renames), or changes caused by other maintenance activities (e.g., bug fixes). This approach allows to differentiate pure-refactoring edits from overlapping non-refactoring edits.

This work is an extension over our previous work [1] with the following novel contributions:

1) We extend the list of supported refactoring types from 15 to 40 (Table 1). The current list covers the majority of the most popular refactoring types applied by developers [34]. To the best of our knowledge, RefactoringMiner 2.0 is the only tool operating at commit-level that supports such low-level refactorings.
2) We support new replacement types (Section 4.3) and heuristics (Section 3.3) to match statements, which improve the quality of the statement mappings produced by our tool, and consequently lead to an increased precision and recall over its predecessor.
3) We address previous limitations, such as the detection of nested refactoring operations (Section 4.4), e.g., when an Extract Method refactoring is applied within the body of another extracted method, and the detection of signature-level refactorings for abstract/interface methods not having a body (Section 4.5).
4) We extend our original refactoring oracle [35] from 3,188 to 7,226 true instances (Section 5.2) to enable the computation of precision and recall for the newly supported refactoring types. These true refactoring instances were found in 536 commits from 185 open-source projects [2], and were validated with multiple tools and experts.
5) We compare the precision and recall of RefactoringMiner 2.0 with its predecessor, and two competitive tools, RefDiff [28], [36] and GumTreeDiff [37], and show that it has a superior accuracy (Sections 5.3 and 5.4).
6) We compare the execution time of RefactoringMiner 2.0 with its predecessor and RefDiff [28], and show that it is much faster (Section 5.6).

2 Related Work

2.1 Refactoring Detection Tools

Demeyer *et al.* [38] proposed the first technique for detecting the refactorings applied between two versions of a software system. They defined heuristics as a combination of change metrics that identify a refactoring type, such as Split/Merge Class, and Move/Split Method. They performed case studies on different versions of three software systems, and concluded that these heuristics are extremely useful in a reverse engineering process, because they reveal where, how, and why an implementation has drifted from its original design.

Antoniol *et al.* [39] used a technique inspired from Information Retrieval (based on Vector Space cosine similarity) to detect discontinuities in classes (e.g., a class replaced with another one, a class split into two, or two classes merged into one). They performed a case study on 40 releases of the dnsjava project, and reported potential refactorings that they detected. Godfrey and Zou [40] implemented a tool that can detect structural refactorings like rename, move, split, and merge for procedural code. They employ origin analysis along with a more expensive call graph analysis to detect and classify these changes. They performed a case study on 12 releases of the PostgreSQL system, and discovered different patterns of code evolution.

Weißgerber and Diehl [41] developed the first technique for the detection of local-scope and class-level refactorings in the commit history of CVS repositories. They first detect refactoring candidates by finding pairs of code elements (i.e., classes, methods, fields) with some differences in their signatures. Next, for each refactoring candidate they use the clone detection tool CCFinder [42] to compare the bodies of the code elements. They configured CCFinder to match code fragments with differences in whitespaces and comments, and with consistently renamed variable identifiers, method names, and references to member names. They manually inspected the commit log messages of two open-source projects to find documented refactorings and compute the recall, and used random sampling to estimate the precision of their approach.

Dig *et al.* [32] developed RefactoringCrawler, which first performs a fast syntactic analysis to find refactoring candidates, and then a precise semantic analysis to find the actual refactorings. The syntactic analysis is based on *Shingles encoding* to find similar pairs of entities (methods, classes, and packages) in two versions of a project. Shingles are "fingerprints" for strings (e.g., method bodies) and enable the detection of similar code fragments much more robustly than the traditional string matching techniques that are not immune to small deviations like renamings or minor edits. The semantic analysis is based on the similarity of *references* (e.g., method calls) to the entities of a candidate refactoring

in the two versions of the project. To compute the recall, the authors manually discovered the applied refactorings in three projects by inspecting their release notes, while they inspected the source code to compute precision.

Xing and Stroulia [43] developed JDEvAn [33], which detects and classifies refactorings based on the design-level changes reported by UMLDiff [44]. UMLDiff is a domain-specific structural differencing algorithm that takes as input two class models of an object-oriented software system, reverse engineered from two corresponding code versions, and automatically detects elementary structural changes on packages, classes, interfaces, fields and methods, based on their name and structure similarity. They evaluated the recall of JDEvAn on two software systems, and found that all documented refactorings were recovered.

Prete *et al.* [45] developed Ref-Finder [31], which detects the largest number of refactoring types (63 of 72) from Fowler's catalog [30], including several low-level refactorings. Ref-Finder encodes each program version using logic predicates that describe code elements and their containment relationships, as well as structural dependencies (i.e., field accesses, method calls, subtyping, and overriding), and encodes refactorings as logic rules. In addition, each refactoring type is encoded as a logic rule, where the *antecedent* predicates represent pre-requisite refactorings or change-facts, and the *consequent* predicate represents a target refactoring type to be inferred. The detection of concrete refactoring instances is achieved by converting the antecedent of each rule into a logic query and invoking the query on the database of logic facts. Some refactoring rules use a special predicate that checks if the word-level similarity between two candidate methods is above a threshold σ. This predicate is implemented as a basic block-level clone detection technique, which removes any beginning and trailing parenthesis, escape characters, white spaces and `return` keywords, and computes word-level similarity between the two code fragments using the longest common subsequence algorithm. Prete *et al.* created a set of correct refactorings by running Ref-Finder with a low similarity threshold ($\sigma = 0.65$) and manually verified them. Then, they computed recall by comparing this set with the results found using a higher threshold ($\sigma = 0.85$) and computed precision by inspecting a sampled data set.

Silva and Valente [28] developed a tool, RefDiff, which takes as input two revisions of a git repository and employs heuristics based on static analysis and code similarity to detect 13 refactoring types. RefDiff represents a source code fragment as a bag of tokens, and computes the similarity of code elements using a variation of the TF-IDF weighting scheme to assign more weight to tokens that are less frequent, and thus have more discriminative power than other tokens. To compute the similarity of fields, RefDiff considers as the "body" of a field, all statements that access this field in the source code of the system. To determine the similarity threshold values for different kinds of code element relationships, the authors applied a calibration process on a randomly selected set of ten commits from ten different open-source projects, for which the applied refactorings are known and have been confirmed by the project developers themselves [2] in order to find the threshold values that yield the best compromise between precision and recall. They evaluated the accuracy of their tool using an oracle of seeded refactorings applied by graduate students in 20 open-source projects.

Silva *et al.* [36] developed RefDiff 2.0, as the first multi-language refactoring detection tool, which relies on the Code Structure Tree (CST) source code representation that abstracts away the specificities of particular programming languages. In its core, RefDiff 2.0, uses the same approach as its predecessor, i.e., it extracts a multiset of tokens for each program element, computes a weight for each token of the source code using a variation of the TF-IDF weighting scheme, and uses a generalization of the Jaccard coefficient, known as weighted Jaccard coefficient, to compute the similarity between two code elements. In contrast to its predecessor that applied a calibration process to determine the similarity threshold values, RefDiff 2.0 uses a single similarity threshold, defined as 0.5 by default, for all kinds of code element relationships. To evaluate the accuracy of the tool on Java projects, they relied on our refactoring oracle [1], [35] constructed from a publicly available dataset of refactoring instances [2], comprising 536 commits from 185 open-source GitHub-hosted projects monitored over a period of two months. For JavaScript and C projects, they manually validated the refactorings detected by the tool to evaluate precision, while they searched for documented refactorings in commit messages to evaluate recall.

A totally different approach to detect refactorings in real-time is to continuously monitor code changes inside the IDE. BeneFactor [46] and WitchDoctor [47] detect manual refactorings in progress and offer support for completing the remaining changes, whereas CodingTracker [34], GhostFactor [48] and ReviewFactor [27] infer fully completed refactorings. While these tools highlight novel usages of fine-grained code changes inside the IDE, RefactoringMiner 2.0 focuses on changes from commits, thus it can be more broadly applied as it is not dependent on an IDE or text editor.

2.2 Limitations

Dependence on Thresholds. The majority of the refactoring detection tools use some kind of similarity metric to measure the code similarity of program elements (e.g,. method declarations) between two software versions or revisions, and use thresholds to determine whether the program elements should be matched, in order to deal with the noise introduced by overlapping refactorings or changes caused by other maintenance activities (e.g., bug fixes) on the same program elements. Typically, each tool provides a set of default threshold values that are empirically determined through experimentation on a relatively small number of projects (one for UMLDiff, three for Ref-Finder and RefactoringCrawler, and ten for RefDiff). The derived threshold values are possibly overfitted to the characteristics of the examined projects, and thus cannot be general enough to take into account all possible ways developers apply refactorings in projects from different domains. As a result, these threshold values might require a calibration to align with the particular refactoring practices applied in a project. This is a tedious task, since it requires an iterative manual inspection of the reported refactorings against the source code to find false positives and re-adjustment of the thresholds.

The problem of deriving appropriate threshold values, in the context of software measurement and metric-based code smell detection, has been extensively investigated by several researchers, who applied various statistical methods and machine learning techniques on a large number of software projects [49], [50], [51], [52], [53]. Dig [54] showed that precision and recall can vary significantly for the same software system based on the selected threshold value. Moreover, Aniche *et al.* [55] has shown that software systems relying on different architectural styles and frameworks require different threshold values for source code metrics. Therefore, experience has shown that it is very difficult to derive *universal* threshold values that can work well for all projects, regardless of their architectural style, application domain, and development practices.

Dependence on Built Project Versions. Several tools require to build the project versions under comparison. For instance, RefactoringCrawler requires method call information to perform semantic analysis, while Ref-Finder and UMLDiff require field access, method call, subtyping and overriding information to extract structural dependencies. This information can be reliably obtained through type, method, and variable bindings resolved by the compiler. A project build can be more easily achieved when comparing released project versions, which are usually shipped with a built binary. However, it can be extremely challenging to build a project when comparing two commits, because only a small portion (38 percent) of the commits can be successfully compiled, and most of them are recent commits [56]. Therefore, the dependence on built project versions is a serious obstacle for performing large-scale refactoring detection in the entire commit history of projects.

Incomplete Oracle. Dig *et al.* [32] created an oracle by inspecting release notes to find information related to refactoring operations. However, Moreno *et al.* [57] manually inspected 990 release notes from 55 open source projects to analyze and classify their content, and found out that only 21 percent of the release notes include refactoring operations, and those are usually general statements specifying the refactored code components (i.e., they lack details about the kinds of refactoring performed). Moreover, in [32], the developers never provided documentation for the "internal" refactorings, but only documented those affecting the public APIs and would therefore be backwards incompatible. Weißgerber and Diehl [41] created an oracle by inspecting commit messages for references to refactoring operations. However, Murphy-Hill *et al.* [29] have shown that developers do not reliably indicate the presence of refactoring in commit log messages. Therefore, an oracle created just by inspecting release notes or commit messages, will probably miss a significant number of undocumented refactorings.

Biased Oracle. Prete *et al.* [45] created an oracle based on the findings of a single tool, namely Ref-Finder, configured with a more relaxed similarity threshold value in order to detect more refactoring instances, and then removed any false positives through manual inspection. Next, they evaluated the precision and recall of the same tool configured with a more strict similarity threshold value. However, this tool might still miss a large number of true instances due to an algorithm design flaw, implementation error, or inappropriate threshold value, leading to a *biased* and *inaccurate* oracle. As a matter of fact, a subsequent independent study has shown that Ref-Finder had an overall precision of 35 percent and an overall recall of 24 percent [58], while a more recent one has shown that Ref-Finder had an overall average precision of 27 percent [59], in contrast to the high precision and recall values (74 and 96 percent, respectively) reported by Prete *et al.* [45] using the oracle explained above. On the other hand, there exist state-of-the-art procedures that use triangulation between multiple tools and experts for determining the ground truth. Such procedures have been used in many software engineering fields, such as design pattern mining [60], [61], [62], to create unbiased benchmarks for evaluating the precision and recall of tools.

Artificial Oracle. Silva and Valente [28] created an oracle by asking graduate students of a Software Architecture course to apply refactorings in open-source projects. This kind of oracle, known as *seeded* refactorings [63], can be used to reliably compute the recall of a tool, since all applied refactorings are known. However, seeded refactorings are not representative of real refactorings for two main reasons. First of all, they are artificial in the sense that they do not necessarily serve an actual purpose (i.e., facilitate a maintenance task, eliminate a code smell, improve code understandability), but are rather random refactoring operations. Second, they are isolated from other maintenance activities, and thus their detection is less challenging. Actual code evolution contains significant noise coming from overlapping changes (e.g., bug fixes, new features, sequential refactoring operations), and thus most real refactorings are not isolated in the commit history of a project. As a matter of fact, Negara *et al.* [64] have shown that 46 percent of refactored program entities are also edited or further refactored in the same commit.

TABLE 2
Limitations of refactoring mining tools

Tool	Input	Detection	Evaluation	Supported Languages
Weißgerber & Diehl [41]	CVS commits ①	CCFinder configuration ③	commit messages ④	Java ⑦
Dig et al. [32]	project versions ②	shingles + references ③	release notes ④	Java ⑦
Xing & Stroulia [33]	project versions ②	UMLDiff configuration ③	change documents ④	Java ⑦
Kim et al. [31]	project versions ②	block-level clone detector ③	single tool ⑤	Java ⑦
Silva & Valente [28]	Git commits ⊖	TF-IDF (variant threshold) ③	seeded refactorings ⑥	Java ⑦
Silva et al. [36]	Git commits ⊖	TF-IDF (fixed threshold) ③	oracle [1] ⊖ commit messages ④	Java, C++ JavaScript

①*obsolete VCS* ②*source code build requirement* ③*similarity threshold definition* ④*incomplete oracle* ⑤*biased oracle* ⑥*artificial oracle* ⑦*language-specific* ⊖*N/A.*

Table 2 provides a comprehensive overview of the limitations of previous refactoring mining tools, with respect to their input, detection method, and evaluation approach. Unlike these previous tools, RefactoringMiner 2.0 neither requires similarity thresholds (that are tedious to calibrate, and might not be generalizable), nor does it require operating on fully built versions of software systems, thus it is applicable in many more contexts. Moreover, whereas

previous tools have been evaluated against 2-3 projects with a small number of refactoring instances (a notable exception is RefDiff, which was evaluated on 20 projects with 448 seeded refactorings), our oracle is orders of magnitude larger comprising 185 projects and 7,226 true refactoring instances. We use triangulation between multiple sources to create one of the most reliable, comprehensive, and representative oracles to date.

3 Source Code Matching

The matching of the statements between two code fragments (i.e., method bodies) is a core function that we use throughout the refactoring detection rules described in Section 4.2. Although this Section focuses on constructs and practices specific to the Java programming language, several concepts, such as the *abstraction* and *argumentization* of statements, and the matching of statements in rounds, can be applied to other programming languages and paradigms.

3.1 Source code representation

The body of a method is represented as a tree capturing the nesting structure of the code, where each node corresponds to a statement, similar to the representation used by Fluri *et al.* [65]. For a composite statement (i.e., a statement that contains other statements within its body, such as `for`, `while`, `do-while`, `if`, `switch`, `try`, `catch`, `synchronized block`, `label`), the node contains the statement's type and the expression(s) appearing within parenthesis before the statement's body. For a leaf statement (i.e., a statement without a body), the node contains the statement itself. Storing AST nodes into memory is not efficient, because a reference to a single AST node results in having the entire CompilationUnit in memory, as AST nodes are linked to their parent nodes. In order to avoid storing AST nodes into memory, for each statement/expression we keep its string representation in a pretty-printed format where all redundant whitespace and multi-line characters are removed. For the pretty-printing of statements/expressions, we rely on Eclipse JDT `ASTNode.toString()`, which uses the internal API `NaiveASTFlattener` to generate the string representation of AST nodes for debug printing. `NaiveASTFlattener` extends the `ASTVisitor` and each overridden `visit()` method uses a standard format to represent the visited AST node type. This means two syntactically identical AST nodes will have the same string representation, as returned by `ASTNode.toString()`, regardless of their differences in whitespace characters. In addition, we use an AST Visitor to extract all variable identifiers, method invocations, class instantiations, variable declarations, types, literals, and operators appearing within each statement/expression and store them in a pretty-printed format within the corresponding statement node. Fig. 1 shows the tree-like representation of the body of method `createAddresses`, along with the information extracted by the AST Visitor for two of its statements.

3.2 Statement Matching

Our statement matching algorithm has been inspired by Fluri *et al.* [65], in the sense that we also match the statements in a bottom-up fashion, starting from matching leaf statements and then proceeding to composite statements. However, in our solution, outlined in Algorithm 1, we do not use any similarity measure to match the statements, and thus we do not require the definition of similarity thresholds.

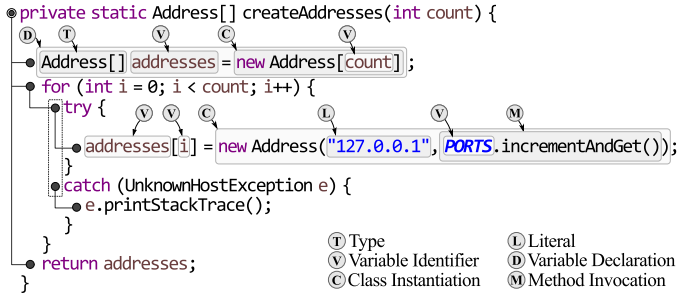

Fig. 1. Representation of a method body as a tree.

To reduce the chances of erroneous matches, we follow a conservative approach, in which we match the statements in rounds, where each subsequent round has a less strict *match condition* than the previous round. Thus, the statements matched in earlier rounds are "safer" matches, and are excluded from being matched in the next rounds. In this way, the next round, which has a more relaxed match condition, has fewer statement combinations to check.

Algorithm 1. Statement matching

Input: Trees T_1 and T_2
Output: Set M of matched node pairs, Sets of unmatched nodes U_{T_1}, U_{T_2} from T_1 and T_2

$M \leftarrow \varnothing, U_{T_1} \leftarrow \varnothing, U_{T_2} \leftarrow \varnothing$
$L_1 \leftarrow T_1$`.leafNodes`, $L_2 \leftarrow T_2$`.leafNodes`
`condition1`$(n_1, n_2) \rightarrow n_1$`.text=`$n_2$`.text` $\wedge$ n_1`.depth =`n_2`.depth`
`condition2`$(n_1, n_2) \rightarrow n_1$`.text=`$n_2$`.text`
`condition3`$(n_1, n_2) \rightarrow$ `replacements`$(n_1$`.text`$, n_2$`.text`$)$
$L'_1, L'_2 =$ `matchNodes`$(L_1, L_2,$ `condition1`$)$ `//round #1`
$L''_1, L''_2 =$ `matchNodes`$(L'_1, L'_2,$ `condition2`$)$ `//round #2`
`matchNodes`$(L''_1, L''_2,$ `condition3`$)$ `//round #3`
$C_1 \leftarrow T_1$`.compositeNodes`, $C_2 \leftarrow T_2$`.compositeNodes`
`condition4` $(n_1, n_2) \rightarrow \exists\ (k_1, k_2) \in M \mid k_1 \in n_1$.children $\wedge$ $k_2 \in n_2$.children
`condition1` $(n_1, n_2) =$ `condition1` $\wedge$ `condition4`
`condition2` $(n_1, n_2) =$ `condition2` $\wedge$ `condition4`
`condition3` $(n_1, n_2) =$ `condition3` $\wedge$ `condition4`
$C'_1, C'_2 =$ `matchNodes`$(C_1, C_2,$ `condition1`$)$ `//round #1`
$C''_1, C''_2 =$ `matchNodes`$(C'_1, C'_2,$ `condition2`$)$ `//round #2`
`matchNodes`$(C''_1, C''_2,$ `condition3`$)$ `//round #3`
$U_{T_1} \leftarrow T_1$`.nodes` $\setminus M_{T_1}, U_{T_2} \leftarrow T_2$`.nodes` $\setminus M_{T_2}$

Function `matchNodes` $(N_1, N_2,$ `matchCondition`$)$
  foreach $n_1 \in N_1$ **do**
    $P \leftarrow \varnothing$
    foreach $n_2 \in N_2$ **do**
      $pn_1, pn_2 \leftarrow$ `preprocessNodes`(n_1, n_2)
      if `matchCondition`(pn_1, pn_2) **then**
        $P \leftarrow P \cup (n_1, n_2)$
    if $|P| > 0$ **then**
      `bestMatch` $\leftarrow$ `findBestMatch`(P)
      $M \leftarrow M \cup$ `bestMatch`
      $N_1 \leftarrow N_1 \setminus$ `bestMatch`$.n_1$
      $N_2 \leftarrow N_2 \setminus$ `bestMatch`$.n_2$
  return N_1, N_2

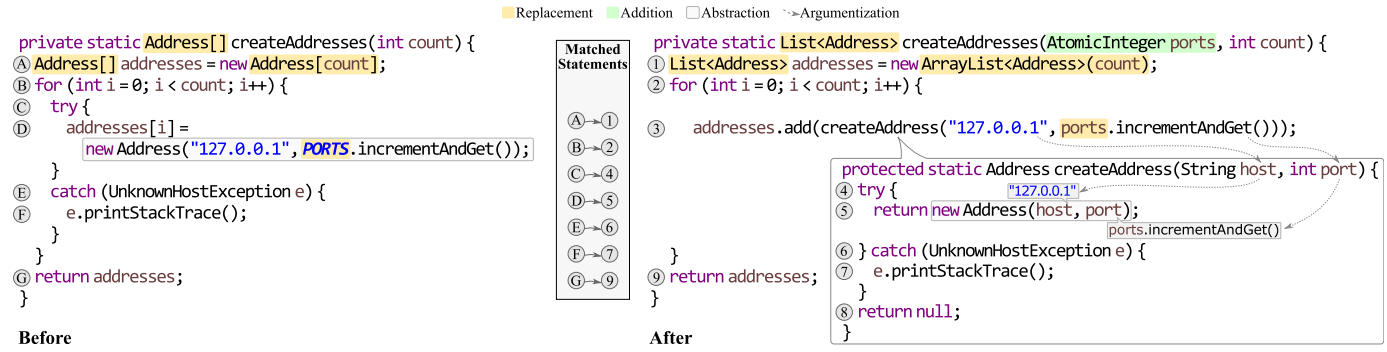

Fig. 2. Statement matching for an EXTRACT METHOD refactoring in https://github.com/hazelcast/hazelcast/commit/76d7f5.

Algorithm 1 takes as input trees T_1 and T_2 representing the nesting structure of the statements inside the body of a method (as shown in Fig. 1) from the parent and child commit, respectively. We match leaf statements in three rounds (lines 2-8). In the first round, we match the statements with identical string representation and nesting depth. In the second round, we match the statements with identical string representation regardless of their nesting depth. In the last round, we match the statements that become identical after replacing the AST nodes being different between the two statements. We match composite statements in three rounds as well (lines 9-16), using exactly the same match conditions as those used for leaf statements combined with an additional condition that requires at least one pair of their children to be matched (line 10), assuming that both composite statements have children.

In all rounds, we apply two pre-processing techniques on the input statements (line 4 in function matchNodes), namely *abstraction* and *argumentization* to deal with specific changes taking place in the code when applying EXTRACT, INLINE, and MOVE METHOD refactorings.

Abstraction. Some refactoring operations, such as EXTRACT and INLINE METHOD, often introduce or eliminate `return` statements when a method is extracted or inlined, respectively. For example, when an expression is extracted from a given method, it appears as a `return` statement in the extracted method. To facilitate the matching of statements having a different AST node type, we *abstract* the statements that wrap expressions. When both statements being compared follow one of the following formats:

- **`expression;`** i.e., expression statement
- `return`**`expression;`** i.e., returned expression
- `Type var =`**`expression;`** i.e., initializer of a variable declaration
- `var =`**`expression;`** i.e., right hand side of an assignment
- `if/while/switch/try(`**`expression`**`)` i.e., condition of a composite statement

then they are abstracted to `expression` before their comparison. Fig. 2 shows an example of abstraction, where the assignment statement Ⓓ from the code before refactoring, and the return statement ⑤ from the code after refactoring, are abstracted to expressions `new Address("127.0.0.1", PORTS.incrementAndGet())` and `new Address(host, port)`, respectively.

Argumentization. Some refactoring operations may replace expressions with parameters, and vice versa. For example, when duplicated code is extracted into a common method, all expressions being different among the duplicated code fragments are *parameterized* (i.e., they are replaced with parameters in the extracted method). The duplicated code fragments are replaced with calls to the extracted method, where each expression being different is passed as an argument. In many cases, the arguments may differ substantially from the corresponding parameter names, leading to a low textual similarity of the code before and after refactoring. Argumentization is the process of replacing parameter names with the corresponding arguments in the code after refactoring. Fig. 2 shows an example of argumentization, where parameter names `host` and `port` are replaced with arguments `"127.0.0.1"` and `ports.incrementAndGet()`, respectively, in statement ⑤.

The same process is applied to the statements of inlined and moved methods. In particular, when an instance method is moved to a target class, we might have a parameter (or a source class field access) of target type that is removed from the original method, or a parameter of source type that is added to the original method. In the case of removal, the removed parameter (or field access) might be replaced with `this` reference in the moved method, while in the case of addition, `this` reference might be replaced with the added parameter in the moved method. By applying *abstraction* and *argumentization*, the original statements Ⓓ and ⑤ in Fig. 2 are transformed to `new Address("127.0.0.1",`**`PORTS`**`.incrementAndGet())` `new Address("127.0.0.1",`**`ports`**`.incrementAndGet())`, respectively, and thus can be identically matched by replacing static field `PORTS` with parameter `ports`. On the other hand, string similarity measures would require a very low threshold to match these statements. For instance, the Levenshtein distance [66] (commonly used for computing string similarity) between the original statements Ⓓ and ⑤ is 44 edit operations, which can be normalized to a similarity of $1 - 44/65 = 0.32$, where 65 is the length of the longest string corresponding to statement Ⓓ. The bigram similarity [67] (used by CHANGEDISTILLER [65]) between statements Ⓓ and ⑤ is equal to 0.3. It is clear that the string similarity measures used by the majority of the refactoring detection tools are susceptible to code changes applied by some refactoring operations, such as parameterization, especially when the arguments differ substantially from the parameter names. Therefore, our pre-processing techniques facilitate the matching of statements with low textual similarity.

Function matchNodes finds all possible matching nodes in tree T_2 for a given node in tree T_1 and stores the matching node pairs into set P. There are certain kinds of statements, such as `return`, `break`, `continue`, assertions in tests, exception throwing, logging, `System.out.print` and other common external API invocations, `try` blocks, and control flow structures, which tend to be repeated in many different parts of a method and are textually identical. In such cases, matchNodes will return multiple matching node pairs. Function findBestMatch(P) (line 8), sorts the node pairs in P and selects the top-sorted one, in order to break the ties when having multiple matching node pairs. Leaf node pairs are sorted based on 3 criteria. First, based on the string edit distance [66] of the nodes in ascending order (i.e., more textually similar node pairs rank higher). Second, based on the absolute difference of the nodes' depth in ascending order (i.e., node pairs with more similar depth rank higher). Third, based on the absolute difference of the nodes' index in their parent's list of children in ascending order (i.e., node pairs with more similar position in their parent's list of children rank higher). Composite node pairs are sorted with an additional criterion, which is applied right after the first criterion: based on the ratio of the nodes' matched children in descending order (i.e., node pairs with more matched children rank higher).

Algorithm 2. Replacements of AST nodes

Input: Statements $s_1$and s_2
Output: Set of syntax-aware replacements

```
1  N_{s1} ← ∅, N_{s2} ← ∅, R ← ∅
2  foreach t ∈ nodeTypes do
3     common_t ← s1.nodes_t ∩ s2.nodes_t
4     N_{s1} ← N_{s1} ∪ { s1.nodes_t \ common_t }
5     N_{s2} ← N_{s2} ∪ { s2.nodes_t \ common_t }
6  d = distance(s1,s2)
7  foreach n_{s1} ∈ N_{s1} do
8     C ← ∅
9     foreach n_{s2} ∈ N_{s2} do
10       if compatibleForReplacement(n_{s1},n_{s2}) then
11          d' = distance(s1.replace(n_{s1},n_{s2}),s2)
12          if d' < d then
13             C ← C ∪ (n_{s1}, n_{s2})
14    if |C| > 0 then
15       best ← smallestDistance(C)
16       d = best.distance
17       r = best.replacement
18       R ← R ∪ r
19       s1 = s1.replace(r.n_{s1},r.n_{s2})
20 if s1 = s2 then
21    return R
22 else
23    return null
```

Algorithm 2 takes as input two statements and performs replacements of AST nodes until the statements become textually identical. This approach has two main advantages over existing methods relying on textual similarity. First, there is no need to define a similarity threshold. There is empirical evidence that developers interleave refactoring with other types of programming activity (e.g., bug fixes, feature additions, or other refactoring operations) [2], [29], [34]. In many cases, the changes caused by these different activities may overlap [64]. Some of these changes may even change substantially the original code being part of a refactoring operation. For example, a code fragment is originally extracted, and then some temporary variables are inlined in the extracted method. The longer the right-hand-side expressions assigned to the temporary variables, the more textually different the original statements will be after refactoring. Therefore, it is impossible to define a *universal* similarity threshold value that can cover any possible scenario of overlapping changes. On the other hand, our approach does not pose any restriction on the replacements of AST nodes, as long as these replacements are syntactically valid. Second, the replacements found within two matched statements can help to infer other edit operations taking place on the refactored code (a phenomenon called *refactoring masking* [68]), such as renaming of variables, generalization of types, and merging of parameters. As a matter of fact, we rely on the replacements returned by Algorithm 2 to infer low-level refactorings that take place within the body of methods (Section 4.3). On the other hand, similarity-based approaches lose this kind of valuable information.

Initially, our algorithm computes the intersection between the sets of *sub-expressions*, such as variable identifiers, method invocations, class instantiations, types, literals, and operators, extracted from each statement, respectively, in order to exclude from replacements the AST nodes being common in both statements, and include only those being different between the statements (lines 2-5). AST nodes that cover the entire statement (e.g., a method invocation followed by `;`) are also excluded from replacements in order to avoid having an excessive number of matching statement combinations. All attempted replacements are *syntax-aware*: 1) We allow only *compatible* AST nodes to be replaced (line 10), i.e., types can be replaced only by types, operators can be replaced only by operators, while all remaining expression types can be replaced by any of the remaining expression types (e.g., a variable can be replaced by a method invocation), 2) We allow replacements of AST nodes having the same *structural properties*, e.g., an argument of a method invocation in the first statement can be replaced with an argument of a method invocation in the second statement. Out of all possible replacements for a given node from the first statement that decrease the original edit distance of the input statements, we select the replacement corresponding to the smallest edit distance (line 15).

In the special case when two method invocation subexpressions are considered for replacement, function compatibleForReplacement(n_{s_1},n_{s_2}) examines the expressions used for invoking the methods (i.e., receiver objects). If these expressions are chains of method invocations, as the case shown in Fig. 3 (commonly known as the *Fluent Interface* [69] pattern in API design), then we extract the individual method invocations being part of each chain and compute their intersection ignoring any differences in the order of the invocations inside each chain. If the number of common invocations is larger than the uncommon ones, then we consider the original method invocations as compatible for

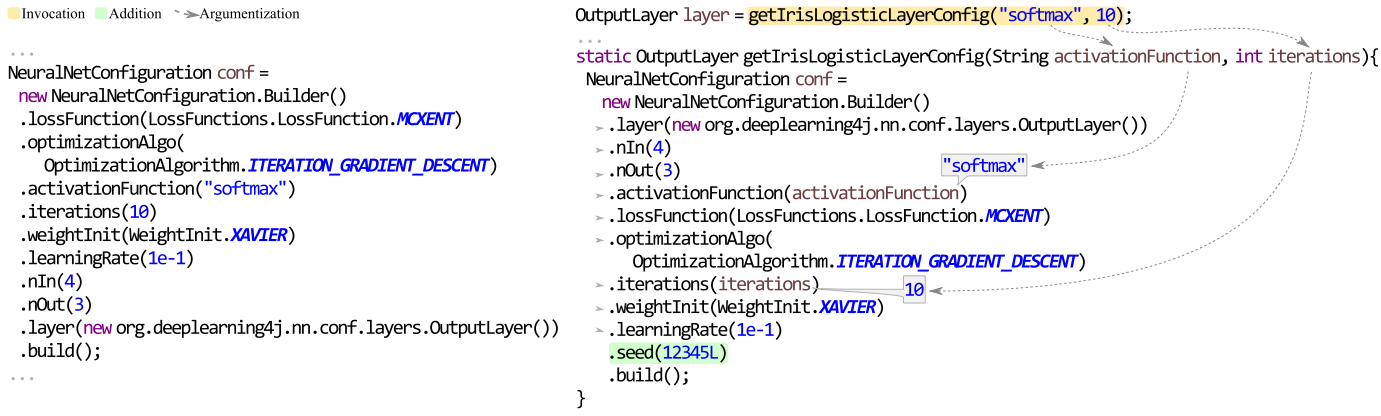

Fig. 3. Method invocation chains following the *Fluent Interface* [69] pattern in https://github.com/eclipse/deeplearning4j/commit/91cdfa.

replacement. In the example of Fig. 3, there are 9 common invocations (two of them are identically matched after applying the argumentization technique), and only 1 uncommon. Notice that string similarity measures produce very low similarity value for this case. For instance, the normalized Levenshtein similarity between the two statements is 0.47, while the bigram similarity is 0.46.

3.3 Matching AST Nodes Covering the Entire Statement

As mentioned before, AST nodes covering the entire statement are excluded from replacements to avoid having an excessive number of matching statements. We consider as AST nodes covering the entire statement, the method invocations and class instance creations matching with **expression** in the statement *abstraction* templates (Section 3.2). Such method invocations and class instance creations might have changes in their list of arguments that cannot be handled by Algorithm 2, such as the insertion or deletion of an argument, and the replacement of multiple arguments with a single one and vice versa. This is because we designed the algorithm to perform only one-to-one AST node replacements and does not support one-to-many, many-to-one, one-to-zero (i.e., deletion), zero-to-one (i.e., insertion) replacements, as this would increase substantially its computational cost. To overcome this limitation, we allow the matching of textually different method invocations and class instance creations covering the entire statement, as long as they satisfy the heuristics shown in Table 3. Heuristics ⑩-⑬ enable matching one-to-one statements with significantly different AST structures, as well as one-to-many statements. Typical AST diff tools, such as GumTreeDiff and ChangeDistiller, either produce very complex edit operation scripts, or fail to match such cases. Similar heuristics are used to match textually different class instance creations, array creations, and super method invocations covering the entire statement.

4 Refactoring Detection

In this section, we present rules for the detection of 40 refactoring types based on the statement mapping information and AST node replacements collected with the algorithms described in Section 3. Although RefactoringMiner 2.0 supports refactorings for Java programs, the detection rules are general enough to be applied to any language following the object-oriented programming paradigm. Moreover, some of the supported refactoring types are common to many different programming paradigms.

4.1 Notation

We adopt and extend the notation defined by Biegel *et al.* [70] for representing the information that we extract from each revision using the Eclipse JDT Abstract Syntax Tree (AST) Parser. Notice that we configure the parser to

TABLE 3
Heuristics for matching textually different method invocations covering the entire statement

Heuristic	Examples of pairs of matched statements
① + ② + ④	`log("Drill Logical",drel);` `log("Drill Logical",drel,logger);`
① + ② + ⑤	`schema.indexCreate(state,labelId,propertyKeyId);` `schema.indexCreate(state,descriptor)`
① + ② + ⑥	`output.push(op);` `output.push(new OperatorNode(op));`
① + ② + ⑦	`connection.getOrConnect(possibleAddress,true);` `connection.getOrConnect(getAddress(),true);`
① + ③ + ⑧	`FunctionResolverFactory.getResolver(castCall);` `FunctionResolverFactory.getExactResolver(castCall);`
② + ③ + ⑨	`attribute.getDefinition().validate(operation,model);` `attribute.validate(operation,model);`
⑩	`this.fileExtension = fileExtension;` `setFileExtension(fileExtension);`

⑪	`new ResponseEnvelope(status,actionResponse,headers);`
	`new Builder().status(status).entity(actionResponse)`
	`.headers(headers).build();`

⑫	`appView = makeWebView();` `appView = appView != null ? appView : makeWebView();`

⑬	`((MeterView)findViewById(R.id.battery))`
	`.setBatteryController(batteryController);`
	`MeterView v = ((MeterView)findViewById(R.id.battery));`
	`v.setBatteryController(batteryController);`

①*identical receiver expression.*
②*identical method invocation name.*
③*identical list of arguments.*
④*argument added/deleted* ⑤*argument split/merged.*
⑥*argument wrapped* ⑦*argument replaced.*
⑧*renamed method invocation* ⑨*different receiver expression* ⑩ *field assignment replaced with setter invocation.*
⑪ *class instance creation replaced with builder call chain.*
⑫ *method invocation replaced with conditional expression.*
⑬ *split/merge invocation to/from multiple statements.*

create the ASTs of the added, deleted, and changed Java compilation units in each revision *without* resolving binding information from the compiler, and thus there is no need to build the source code. Consequently, all *referenced types* (e.g., parameters, variable/field declarations, extended superclass, implemented interfaces) are stored as they appear in the AST, as we are not able to obtain their fully qualified names. For each revision r, we extract the following information:

- TD_r: The set of type declarations (i.e., classes, interfaces, enums) affected in r. For a child commit, this set includes the type declarations inside changed and added Java files, while for a parent commit, this set includes the type declarations inside changed and removed Java files. Each element td of the set is a tuple of the form (p, n, F, M), where p is the parent of td, n is the name of td, F is the set of fields declared inside td, and M is the set of methods declared inside td. For a top-level type declaration p corresponds to the package of the compilation unit td belongs to, while for a nested/inner type declaration p corresponds to the package of the compilation unit td belongs to concatenated with the name of the outer type declaration under which td is nested.
- F_r: The set of fields inside the type declarations of TD_r. It contains tuples of the form (c, t, n), where c is the fully qualified name of the type declaration the field belongs to (constructed by concatenating the package name p with the type declaration name n), t is the type of the field, and n is the name of the field.
- M_r: The set of methods inside the type declarations of TD_r. It contains tuples of the form (c, t, n, P, b), where c is the fully qualified name of the type declaration the method belongs to, t is the return type of the method, n is the name of the method, P is the ordered parameter list of the method, and b is the body of the method (could be `null` if the method is abstract or native).
- D_r: The set of directories containing the modified Java files in commit r, extracted from the Git tree object representing the hierarchy between files. Each directory is represented by its path p.

4.2 Refactoring Detection Based on Matched Statements

The detection of refactorings takes place in two phases. The first phase is less computationally expensive, since the code elements are matched only based on their signatures. Our assumption is that two code elements having an identical signature in two revisions correspond to the same code entity, regardless of the changes that might have occurred within their bodies. The second phase is more computationally expensive, since the remaining code elements are matched based on the statements they have in common within their bodies. In a nutshell, in the first phase, our algorithm matches code elements in a top-down fashion, starting from classes and continuing to methods and fields. Two code elements are matched only if they have an identical *signature*. Assuming a and b are two revisions of a project:

- Two type declarations td_a and td_b have an identical signature, if $td_a.p = td_b.p \ \wedge td_a.n = td_b.n$
- Two fields f_a and f_b have an identical signature, if $f_a.c = f_b.c \ \wedge f_a.t = f_b.t \ \wedge f_a.n = f_b.n$
- Two methods m_a and m_b have an identical signature, if $m_a.c = m_b.c \ \wedge m_a.t = m_b.t \ \wedge m_a.n = m_b.n \ \wedge m_a.P = m_b.P$
- Two directories d_a and d_b are identical, if $d_a.p = d_b.p$

After the end of the first phase, we consider the unmatched code elements from revision a as *potentially deleted*, and store them in sets TD^-, F^-, M^-, and D^-, respectively. We consider the unmatched code elements from revision b as *potentially added*, and store them in sets TD^+, F^+, M^+, and D^+, respectively. Finally, we store the pairs of matched code elements between revisions a and b in sets $TD^=$, $F^=$, $M^=$, and $D^=$, respectively.

In the second phase, our algorithm matches the remaining code elements (i.e., the *potentially deleted* code elements with the *potentially added* ones) in a bottom-up fashion, starting from methods and continuing to classes, to find code elements with signature changes or code elements involved in refactoring operations.

Examination Order of Refactoring Types. We detect the refactoring types in the order they appear in Table 4 by applying the rules shown in the second column of the table. The order of examination is very important for the accuracy of our approach. We order the refactoring types according to their *spatial locality* with respect to moving code, starting from refactorings that do not move code (i.e., refactorings changing the signature of a method/type declaration), then proceeding with refactorings that move code locally within the same container (i.e., extracting/inlining a local method), followed by refactorings that move code between existing containers (i.e., moving a method to another existing class), and ending with refactorings that move code to new containers (i.e., moving methods to a newly extracted class, or moving classes to a newly introduced package). The intuition behind this order comes from empirical evidence showing that small and local refactorings are more frequent than big and global ones [34], [71], and thus there is a higher probability that the potentially added/deleted code elements resulted from local rather than global refactorings. Whenever a refactoring type is processed, we remove the matched code elements from the sets of potentially deleted/added code elements, and add them to the corresponding sets of matched code elements. This decreases the number of code elements examined in the refactoring types that follow, thus reducing the noise level and improving accuracy. For example, by matching the pairs of methods with changes in their signature, we exclude these methods from being considered as sources or targets of Move Method refactorings, and also enable the detection of extracted/inlined methods from/to methods with changes in their signature. By finding the locally extracted/inlined methods, we exclude them from being considered as sources or targets of Move Method refactorings. By finding the moved/renamed classes, we exclude the methods/fields they contain from being considered as sources or targets of Move Method/Field refactorings. As we discuss in the evaluation (Section 5.3), missing the detection of moved/renamed classes can lead to the erroneous detection of the methods/fields they contain as being moved, which affected the precision of RefDiff 0.1.1.

TABLE 4
Refactoring detection rules

Refactoring type	Rule
Change Method Signature m_a to m_b	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_a.b, m_b.b)\ \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge$ ① $(U_{T_1} = \varnothing \wedge U_{T_2} = \varnothing \wedge \texttt{allExactMatches}\,(M)) \vee$ ② $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}\,(m_a, m_b) \wedge \texttt{compatibleSignatures}\,(m_a, m_b)) \vee$ ③ $(\lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}\,(m_a, m_b) \wedge \exists\,\texttt{extract}\,(m_a, m_x)) \vee$ ④ $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \texttt{locationHeuristic}\,(m_a, m_b) \wedge \exists\,\texttt{inline}\,(m_x, m_b))$ $m_a.n \neq m_b.n \Rightarrow$ RENAME METHOD $m_a.t \neq m_b.t \Rightarrow$ CHANGE RETURN TYPE
Change Class Signature td_a to td_b	$\exists\,(td_a, td_b) \mid td_a \in TD^- \wedge td_b \in TD^+ \wedge (td_a.M \supseteq td_b.M \vee td_a.M \subseteq td_b.M) \wedge (td_a.F \supseteq td_b.F \vee td_a.F \subseteq td_b.F)$ $td_a.p \neq td_b.p \Rightarrow$ MOVE CLASS $td_a.n \neq td_b.n \Rightarrow$ RENAME CLASS $td_a.p \neq td_b.p \wedge td_a.n \neq td_b.n \Rightarrow$ MOVE & RENAME CLASS
Extract Method m_b from m_a	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_a.b, m_b.b)\ \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge$ $\neg\texttt{calls}\,(m_a, m_b) \wedge \texttt{calls}\,(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert$
Inline Method m_b to $m_{a'}$	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_b.b, m_{a'}.b)\ \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^- \wedge m_{a'}.c = m_b.c \wedge$ $\texttt{calls}\,(m_a, m_b) \wedge \neg\texttt{calls}\,(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_1}\rvert$
Move Method m_a to m_b	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_a.b, m_b.b)\ \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c\ \wedge \lvert M\rvert > \lvert U_{T_1}\rvert\ \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge$ $(td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge m_b \in td_{b'} \wedge$ $(\texttt{importsType}\,(td_{a'}, m_b.c) \vee \texttt{importsType}\,(td_b, m_a.c))$ $\texttt{subType}\,(m_a.c, m_b.c) \Rightarrow$ PULL UP METHOD $\texttt{subType}\,(m_b.c, m_a.c) \Rightarrow$ PUSH DOWN METHOD
Move Field f_a to f_b	$\exists\,(f_a, f_b) \mid f_a \in F^- \wedge f_b \in F^+ \wedge f_a.c \neq f_b.c \wedge f_a.t = f_b.t \wedge f_a.n = f_b.n \wedge$ $(td_a, td_{a'}) \in TD^= \wedge f_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge f_b \in td_{b'} \wedge$ $(\texttt{importsType}\,(td_{a'}, f_b.c) \vee \texttt{importsType}\,(td_b, f_a.c))$ $\texttt{subType}\,(f_a.c, f_b.c) \Rightarrow$ PULL UP FIELD $\texttt{subType}\,(f_b.c, f_a.c) \Rightarrow$ PUSH DOWN FIELD
Extract m_b from m_a & Move to $m_b.c$	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_a.b, m_b.b)\ \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c \wedge$ $\neg\texttt{calls}\,(m_a, m_b) \wedge \texttt{calls}\,(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge (td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge \texttt{importsType}\,(td_{a'}, m_b.c)$
Move m_b to $m_{a'.c}$ & Inline to $m_{a'}$	$\exists\,(M, U_{T_1}, U_{T_2}) = \texttt{matching}\,(m_b.b, m_{a'}.b)\ \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^- \wedge m_{a'}.c \neq m_b.c \wedge$ $\texttt{calls}\,(m_a, m_b) \wedge \neg\texttt{calls}\,(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_1}\rvert \wedge (td_a, td_{a'}) \in TD^= \wedge m_{a'} \in td_{a'} \wedge \texttt{importsType}\,(td_a, m_b.c)$
Extract Supertype td_b from td_a	$\exists\,(td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \wedge td_b \in TD^+ \wedge \texttt{subType}\,(\texttt{type}\,(td_{a'}), \texttt{type}\,(td_b))$ $\exists\,\texttt{pullUp}\,(m_a, m_b)\ \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\,\texttt{pullUp}\,(f_a, f_b)\ \mid f_a \in td_a \wedge f_b \in td_b \Rightarrow$ EXTRACT SUPERCLASS $\exists\,(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \wedge \texttt{identicalSignatures}\,(m_a, m_b) \wedge m_b.b = \texttt{null} \Rightarrow$ EXTRACT INTERFACE
Extract Subtype td_b from td_a	$\exists\,(td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \wedge td_b \in TD^+ \wedge \texttt{subType}\,(\texttt{type}\,(td_b), \texttt{type}\,(td_{a'}))$ $\exists\,\texttt{pushDown}\,(m_a, m_b)\ \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\,\texttt{pushDown}\,(f_a, f_b)\ \mid f_a \in td_a \wedge f_b \in td_b \Rightarrow$ EXTRACT SUBCLASS
Extract Type td_b from td_a	$\exists\,(td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \wedge td_b \in TD^+ \wedge \neg\texttt{subType}\,(\texttt{type}\,(td_{a'}), \texttt{type}\,(td_b)) \wedge$ $\neg\texttt{subType}\,(\texttt{type}\,(td_b), \texttt{type}\,(td_{a'})) \wedge \texttt{importsType}\,(td_{a'}, \texttt{type}\,(td_b))$ $\exists\,\texttt{move}\,(m_a, m_b)\ \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\,\texttt{move}\,(f_a, f_b)\ \mid f_a \in td_a \wedge f_b \in td_b \Rightarrow$ EXTRACT CLASS
Change Package p_a to p_b	$\exists\,(p_a, p_b) \mid \texttt{path}\,(p_a) \in D^- \wedge \texttt{path}\,(p_b) \in D^+ \wedge \exists\,\texttt{MoveClass}\,(td_a, td_b)\ \mid td_a.p = p_a \wedge td_b.p = p_b$

$\texttt{matching}(T_1, T_2)$ returns a set of matched statement pairs (M) between the trees T_1 and T_2 representing method bodies, and two sets of unmatched statements from T_1 (U_{T_1}) and T_2 (U_{T_2}), respectively $\texttt{indexOf}(m, td)$ returns the position of m inside type declaration td $\texttt{typeDecl}(c)$ returns the type declaration of type c $\texttt{type}(td)$ returns the qualified name of type declaration td $\texttt{locationHeuristic}(m_a, m_b) = \lvert indexOf(m_a, typeDecl(m_a.c)) - indexOf(m_b, typeDecl(m_b.c))\rvert \leq \lvert M_c^- - M_c^+\rvert$ $\texttt{importsType}(td, t)$ returns true if type declaration td depends on type t $\texttt{calls}(m_a, m_b)$ returns true if method m_a calls directly or indirectly m_b $\texttt{compatibleSignatures}(m_a, m_b) = m_a.P \supseteq m_b.P \vee m_a.P \subseteq m_b.P \vee \lvert m_a.P \cap m_b.P\rvert \geq \lvert(m_a.P \cup m_b.P) \setminus (m_a.P \cap m_b.P)\rvert$ $\texttt{subType}(c_a, c_b)$ returns true if c_a is a direct or indirect subclass of c_b or implements interface c_b $\texttt{path}(p)$ returns the directory path for package p.

Previous works have also investigated the detection order of different refactoring types. In REF-FINDER (Prete *et al*. [45]), the refactoring detection rules have a topological order, because the logical queries used to detect some composite refactorings depend on other prerequisite basic refactorings. However, Prete *et al*. do not discuss about the detection order of basic refactoring types that do not depend on others. Our refactoring detection rules are independent from each other, with the exception of CHANGE PACKAGE, which depends on previously detected MOVE CLASS refactorings. We also have composite refactorings, which are composed from basic ones. For example, EXTRACT CLASS is composed of MOVE METHOD/FIELD refactorings. However, these are moves to newly added classes in the child commit (i.e., the extracted classes), while the refactorings detected by the MOVE METHOD/FIELD rules involve only methods/fields moved between previously existing classes in the parent commit. In REFACTORINGCRAWLER (Dig *et al*. [32]), the refactoring detection rules are applied in a top-down fashion following the containment relationship between program elements. For example, the detection rules first establish renamings on packages, then on classes within the matched packages, and finally on methods within the matched classes. However, this order requires to run each detection strategy multiple times, since the detection of refactorings on child program elements may enable the discovery of new refactorings on their parent container that could not be established in the previous run. Obviously, the repeated execution of the detection rules may introduce a considerable computation overhead. In our approach, each detection rule is executed only once by ordering the refactoring types based on their *spatial locality*.

When comparing two sets of potentially added/deleted code elements, we always place the set with smaller cardinality in the outer loop, and the set with larger cardinality in the inner loop. In this way, we avoid "forced" matches of code elements, which might occur if some elements of the smaller set are *actually* added/deleted, and thus have no *true match* in the larger set. Having the larger set in the outer

loop might result in matching pairs of actually added/deleted elements, which marginally pass the rules shown in Table 4.

Best Match selection. For the refactoring types involving statement matching in their detection rule, when a code element (i.e., method declaration) has multiple matches, we always select the best match. The reason is that the same method declaration signature cannot be part of multiple refactoring operations. For example, a method cannot be renamed to multiple methods. However, parts of the method declaration's body can participate in other refactoring operations, such as EXTRACT METHOD or INLINE METHOD. A notable exception is PUSH DOWN refactoring, where it is possible that the same method or field is pushed down from the superclass to multiple subclasses. Our algorithm sorts the matching method pairs based on 4 criteria, which serve as proxies for method similarity at statement level. First, based on the total number of matched statements in descending order (i.e., method pairs with more matched statements rank higher). Second, based on the total number of exactly matched statements in descending order (i.e., method pairs with more identical statements rank higher). Third, based on the total edit distance [66] between the matched statements in ascending order (i.e., method pairs with more textually similar statements rank higher). Fourth, based on the edit distance between the method names in ascending order (i.e., method pairs with more textually similar names rank higher).

As Table 4 shows, the refactoring types examined first have more elaborate and strict rules. This is crucial to avoid early erroneous matches that would negatively affect the accuracy of the detected instances for the refactoring types that follow. For example, the *location heuristic* applied in sub-rules ②, ③, and ④ of the Change Method Signature refactoring type, ensures that the positional difference of two matched methods is less or equal to the absolute difference in the number of methods added to and deleted from a given type declaration. The intuition behind this heuristic is that developers do not tend to change the position of an already existing method inside its type declaration when changing its signature. Assuming that only method renames take place in a type declaration, the number of potentially added and deleted methods will be equal, and thus the location heuristic will be satisfied only for the method pairs having the same position before and after refactoring. This heuristic is particularly effective in cases of extensive method signature changes in test classes (e.g., see the case of extensive unit test renames in project cassandra[1]), where developers tend to copy-and-modify older unit tests to create new ones [72], and thus several methods share very similar statements with each other. Sub-rules ③ and ④ take into account the case where a method with a signature change has a significant portion of its body extracted or inlined, respectively. For instance, in the case shown in Fig. 2, the result of statement matching between methods `createAddresses` before and after refactoring is $M = \{(A,1),(B,2),(G,9)\}$, i.e., $|M| = 3$, while $U_{T_1} = \{C,D,E,F\}$, i.e., $|U_{T_1}| = 4$, and $U_{T_2} = \{3\}$, i.e., $|U_{T_2}| = 1$, and thus sub-rule ② fails to match the methods.

1. https://github.com/apache/casandra/commit/446e25#diff-8d5005607847694afae01a22fa8fdbce

TABLE 5
Supported replacement types within matched statements

AST node	Replacement	Example
Variable	Variable	`filePath→file`
	Array access	`paras.add(id)→paras.add(id[0])`
	Invocation	`rdbms.setconf(conf)→` `rdbms.get().setConf(conf)`
	Literal	`Joiner.on("\n")→Joiner.on(newLine)`
Method Invocation	Invocation	`p.newChildbuilder().executor(e).build()→` `p.newChildbuilder().build()`
	Variable	`return Arrays.asList(files);→` `return files;`
	Literal	`map.put(nameSpace,call.isSourceFile())→` `map.put(nameSpace,false)`
Literal	Number	`3→3L`
	String	`"1"→"1.0"`
Class Instance Creation	Type	`new ArrayList<>()→new HashSet<>()`
	Array creation	`new ArrayList<>(len)→` `new H1SampleWrapper[len]`
	Invocation	`f = new File(root,path);→` `f = root.resolve(path);`
Type	Type	`int i = 0;→byte i = pos;`
Operator	Prefix	`if(passes.isEmpty())→if(!passes.isEmpty())`
	Infix	`period != null→period == null`
Operand	Add/Delete	`String s = "[" + getID() + "]";↔` `String s = getType() + "[" + getID() + "]";`
Argument List	Split/Merge	`functionType(returnType,parameters) ↔` `functionType(returnType,required,optional)`
	Add/Delete	`s = save(outStream,flags)↔` `s = save(outStream,flags,pswrd)`
	Wrap	`output.push(op);→` `output.push(new OperatorNode(op));`

On the other hand, sub-rule ③ matches successfully the methods, because $|M| > |U_{T_2}|$ and there exists at least one method extracted from the original `createAddresses`.

4.3 Refactoring Detection Based on Replacements

At the end of the process explained in Section 4.2, we obtain statement mappings between the pairs of matched methods with identical or changed signatures, between the pairs of moved methods, and between the methods extracted/inlined from/to other methods. Each one of the obtained statement mappings may contain AST node replacements that were performed to make the statements textually identical (Algorithm 2). We rely on these replacements to infer low-level refactorings that take place within the body of methods. Table 5 shows an overview of the replacement types that are currently supported by REFACTORINGMINER 2.0.

Detection of Renamed Variables. To detect variable renames we utilize all replacements of `Variable→Variable` type found in matched statements present within the scopes of the two variables. The scope of a *local variable* extends from its declaration statement until the end of the parent block in which it is declared. The scope of a *parameter* is the entire body of the method in which it is declared. The scope of a *field* is the entire type declaration in which it is declared, as well as the type declarations inheriting the field's type declaration, if the field is not declared as `private`.

To report that variable x is renamed to y, we perform the following *consistency* checks:

1) For all replacements found in matched statements within the scopes of variables x and y, if x appears on the left side, y should appear on the right side and vice versa.
2) No reference of x should appear in statements within the scope of y in the child commit.
3) No reference of y should appear in statements within the scope of x in the parent commit.
4) If x and y are local variables, there should exist a mapping containing their declaration statements.
5) Assuming x is a local variable declared inside method m in the parent commit, its declaration should not appear inside a method extracted from m in the child commit.

The last check ensures that variable x is not erroneously reported as renamed, because of new statements added in method m that reference variable y and are coincidentally matched with statements referencing x.

Based on the kinds of variables x and y, a different refactoring type is reported:

1) Both x and y are local variables $\Rightarrow$ Rename Variable
2) Both x and y are parameters $\Rightarrow$ Rename Parameter
3) x is not parameter and y is parameter $\Rightarrow$ Parameterize Variable
4) x is not field and y is field $\Rightarrow$ Replace Variable with Field
5) Both x and y are fields declared in td_x, td_y, respectively
   a) If $\exists\ (td_x, td_y) \in TD^{=} |\ x \in F^{-} \wedge y \in F^{+} \Rightarrow$ Rename Field
   b) If $\exists\ (td_x, td_y) \in TD^{=} |\ x \in F^{=} \wedge y \in F^{+} \Rightarrow$ Replace Field
   c) If $\nexists\ (td_x, td_y) \in TD^{=} \Rightarrow$ Move and Rename Field

Our approach for detecting variable renames is based on the *references* of the variables. However, there are some cases where it is not possible to obtain references for a variable, e.g., if parameter x of an abstract method declared in an interface or an abstract class is renamed to y, because abstract methods do not have a body. To overcome this limitation, we examine if a similar parameter rename x→y occurred in any of the overriding methods having a body. In this way, we indirectly infer parameter renames in abstract methods.

Detection of Extracted Variables. To detect extracted variables we utilize all replacements of *→Variable type, where * can be a method invocation, class instance creation, or literal expression, found in matched statements present within the scope of the variable on the right hand side of each replacement.

To report that variable y is extracted, we perform the following checks:

1) If y is a local variable, it should be a newly added variable in the child commit.
2) If y is a field, it should be a newly added field in the child commit.
3) The initializer of y should be textually identical with the expression on the left hand side of a replacement within the scope of y, tolerating any overlapping refactorings that affect the expression, such as variable or method renames.

Detection of Inlined Variables. To detect inlined variables we utilize all replacements of Variable→* type, where * can be a method invocation, class instance creation, or literal expression, found in matched statements present within the scope of the variable on the left hand side of each replacement.

To report that variable x is inlined, we perform the following checks:

1) If x is a local variable, it should be a deleted variable from the parent commit.
2) If x is a field, it should be a deleted field from the parent commit.
3) The initializer of x should be textually identical with the expression on the right hand side of a replacement within the scope of x, tolerating any overlapping refactorings that affect the expression, such as variable or method renames.

Detection of Merged Variables. To detect merged variables we utilize all replacements of Set<Variable>→Variable type (i.e., merge of $n > 1$ variables, appearing as arguments, to a single variable), and all replacements of Variable→Variable.Invocation, found in matched statements within the scope of the variables on the left and right hand side of each replacement. Variable.Invocation refers only to cases where the Variable is a Parameter Object [30], and Invocation is a call to a getter method. In many cases, merged variables are extracted into a class and become fields accessed through getter methods.

To report that a set X of variables $\{x_1, .., x_n\}$ is merged to variable y, we perform the following checks:

1) All variables/parameters/fields in X should be deleted from the parent commit.
2) Variable/Parameter/Field y should be a newly added variable in the child commit.
3) For all replacements found in matched statements within the scopes of the variables in set X and variable y, if X appears on the left side, y should appear on the right side and vice versa.
4) All replacements of Variable→Variable.Invocation type should follow the pattern x_i→y.getter, where $x_i \in$ X and getter is a call to the getter method returning the field corresponding to x_i.

Detection of Split Variables. To detect split variables we utilize all replacements of Variable→Set<Variable> type (i.e., split of a single variable to $n > 1$ variables appearing as arguments), and all replacements of Variable.Invocation→Variable, found in matched statements within the scope of the variables on the left and right hand side of each replacement.

To report that a variable x is split to a set Y of variables $\{y_1, .., y_n\}$, we perform the following checks:

1) Variable/Parameter/Field x should be deleted from the parent commit.
2) All variables/parameters/fields in Y should be newly added in the child commit.
3) For all replacements found in matched statements within the scopes of variable x and the variables in

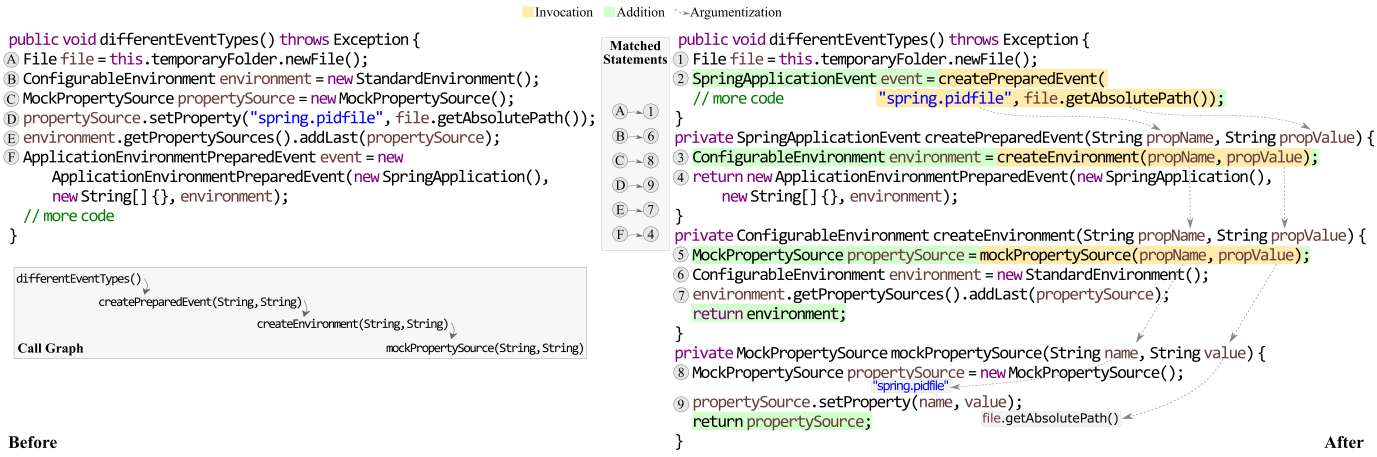

Fig. 4. Nested Extract Method refactorings in https://github.com/spring-projects/spring-boot/commit/becce.

set Y, if x appears on the left side, Y should appear on the right side and vice versa.

4) All replacements of `Variable.Invocation→Variable` type should follow the pattern `x.getter→`y_i, where $y_i \in$ Y and `getter` is a call to the getter method returning the field corresponding to y_i.

Detection of Variable Type Changes. To detect changes in the type of local variables we utilize all replacements of `Type→Type` type. Such replacements occur in matched variable declaration statements, in the parameter of matched `enhanced-for` loops, in the initializer(s) of matched `for` loops, in the exception of matched `catch` clauses, and in the resource(s) of matched `try` statements. We report a Change Variable Type refactoring, if the names of the variables related to the type change are the same, or these variables are involved in a Rename Variable refactoring.

4.4 Nested Refactoring Operations

A refactoring operation that takes place in code resulting from the application of another refactoring operation is a *nested refactoring*. For example, the renaming or extraction of local variables inside the body of an extracted method are nested refactoring operations. Such low-level nested refactorings are detected using the same rules defined in Section 4.3 based on the replacements found in the statements of extracted methods. However, the detection of nested Extract Method refactoring operations is more challenging for two reasons. First, the call sites of the nested extracted methods are not present in the original method from which the code was extracted, but are dispersed in multiple different methods. The rule defined in Table 4 assumes only direct calls to extracted methods inside the body of the original method. Second, the code extracted from the original method needs to be matched with the code of multiple methods (i.e., the directly and nested extracted methods).

Fig. 4 shows a real case of nested Extract Method refactorings found in project spring-boot. By observing the call graph shown at the bottom-left of Fig. 4, there are 3 levels of nested extracted methods. Each extracted method calls the subsequent one, until we reach method `mockPropertySource()`. The developer performed these code extractions to reuse methods `createPreparedEvent()`, `createEnvironment()` and `mockPropertySource()` and at the same time eliminate duplicated code existing in method `overridePidFileWithSpring()`.

To enable the detection of nested Extract Method refactorings, we first construct a partial call tree of the original method in the child commit (i.e., $m_{a'}$) that includes only newly added methods in the child commit, which have not been matched with previously existing ones in the parent commit (i.e., $m_{b^*} \in M^+$). Next, we traverse the call tree in breadth-first order and attempt to match the statements of the original method (m_a) with the statements of the currently visited method in the call tree (m_{b^*}). If the conditions of the Extract Method rule defined in Table 4 hold, then we report that m_{b^*} was extracted from m_a.

In the example of Fig. 4, the first method in the partial call tree is `createPreparedEvent()`. The result of statement matching between methods `differentEventTypes()` and `createPreparedEvent()` is $M = \{(F, 4)\}$, i.e., $|M| = 1$, and $U_{T_2} = \{3\}$, i.e., $|U_{T_2}| = 1$. We ignore the unmatched statement from `createPreparedEvent()`, because it contains a call to `createEnvironment()`, which is a subsequent method in the partial call tree. This unmatched statement did not originally exist and it is actually introduced from a nested Extract Method refactoring. Next, we match the statements between methods `differentEventTypes()` and `createEnvironment()`. The result is $M = \{(B, 6), (E, 7)\}$, i.e., $|M| = 2$, and $U_{T_2} = \{5\}$, i.e., $|U_{T_2}| = 1$. Again, the unmatched statement from `createEnvironment()` is ignored, because it contains a call to `mockPropertySource()`, which is a subsequent method in the partial call tree. Finally, we match the statements between methods `differentEventTypes()` and `mockPropertySource()`. It should be emphasized that the argumentization process is propagated throughout the call tree traversal, which means that parameter `name` is replaced with `"spring.pidfile"` and parameter `value` is replaced with `file.getAbsolutePath()`. Therefore, the matching result is $M = \{(C, 8), (D, 9)\}$, i.e., $|M| = 2$, and $U_{T_2} = \{\}$, i.e., $|U_{T_2}| = 0$. In all 3 examined methods in the partial call tree, the rule $|M| > |U_{T_2}|$ holds, and thus we report that the 3 methods are extracted from `differentEventTypes()`.

As shown in Table 6, in our dataset of 536 commits from 185 projects, we found 8 commits from 8 different projects with nested Extract Method, and 2 commits from 2 different projects with nested Inline Method refactorings. The

TABLE 6
Commits with nested refactorings

Refactoring Type	Commit
Extract Method	https://github.com/spring-projects/spring-boot/commit/becce
	https://github.com/skylot/jadx/commit/2d8d
	https://github.com/belaban/JGroups/commit/f1533
	https://github.com/facebook/buck/commit/7e104c
	https://github.com/google/closure-compiler/commit/ea96643
	https://github.com/checkstyle/checkstyle/commit/5a9b72
	https://github.com/google/j2objc/commit/d05d9
	https://github.com/infinispan/infinispan/commit/043030
Inline Method	https://github.com/wildfly/wildfly/commit/4aa2e8
	https://github.com/jfinal/jfinal/commit/881b

percentage of commits having nested method extractions/inlines is small (5 percent) in our dataset, which includes commits between June 8^{th} and August 7^{th} 2015. However, we assume that nested refactorings occur more frequently when *squashing* commits (git squash converts a series of commits into a single commit), because the longer the commit sequence for a given maintenance task, the larger the probability of performing overlapping edits, and thus overlapping refactorings. *Squashing* is a highly promoted and adopted practice [73], [74], because it improves the quality of the change history. Developers tend to squash all commits in a pull request before merging to the development branch.

4.5 Refactoring Inference

An inherent limitation of our approach is its inability to detect signature-level refactorings for methods not having a body (i.e., abstract and interface methods), since our refactoring detection rules rely on statement mapping information. To overcome this limitation, we infer refactorings for methods not having a body from already detected refactoring operations on methods having identical signatures with abstract and interface methods. Our intuition is that a change in the signature of an abstract or interface method should propagate to all concrete implementations of that method (i.e., overriding methods) to ensure that the committed code can be compiled successfully. For every combination of abstract/interface method pairs inside matched type declarations in $TD^{=}$, we search for pairs of already matched method declarations in $M^{=}$ having identical signatures, and their container type declaration extends/implements the type containing the abstract/interface method pair. If the matching method pairs are involved in Rename Method, Change Return Type, Rename Parameter, Change Parameter Type refactoring types, and the same change appears in the abstract/interface method pair, then we match the pair of abstract/interface methods and report the corresponding refactorings.

Finally, we use type change information to infer refactorings for renamed classes that could not be matched by the rule described in Table 4. Our intuition is that a rename of a local type declaration should propagate to all references of this type to ensure that the committed code can be compiled successfully. First, we extract recurring type changes (two or more occurrences) from already detected type-related refactoring operations on variables, parameters, fields and method return types. For each type change pattern $T_1 \rightarrow T_2$, we search if there exist a deleted type declaration in TD^{-} named T_1 and an added type declaration in TD^{+} named T_2. If such a pair of matching type declarations is found, then we report the corresponding Rename Class refactoring.

5 Evaluation

The two main criteria to evaluate a refactoring mining tool is *accuracy* and *execution time*. A high accuracy will increase the reliability of studies collecting refactoring operations from the commit history of projects to investigate various software evolution phenomena, as well as the effectiveness of tools that depend on refactoring information, or are affected by refactoring noise. A fast execution time will allow to create larger refactoring datasets to strengthen the validity of empirical studies or train learning-based refactoring recommendation systems, and enable novel applications of refactoring mining at commit time.

In information retrieval, accuracy is typically computed by measuring precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$), where TP is the number of true positives (i.e., valid refactoring operations mined by a tool), FP is the number of false positives (i.e., invalid refactoring operations reported by a tool), and FN is the number of false negatives (i.e., valid refactoring operations missed by a tool). True and false positives can be measured through manual validation, i.e., experts can inspect the refactoring operations reported by a tool and determine whether they are valid or not. However, to measure false negatives, there should exist a ground truth of all true positives. As discussed in Section 2, there are many challenges to create a ground truth of refactoring operations at commit level, because the developers rarely or vaguely mention their refactoring activity in commit messages, release notes, and changelogs.

In our previous work [1], we created an oracle by considering as ground truth the union of valid refactoring operations reported by two tools, namely RefactoringMiner 1.0 and RefDiff 0.1.1 [28]. We selected RefDiff, because it is the only other refactoring mining tool operating at commit level, and it outperforms other widely used refactoring detection tools, such as Ref-Finder and RefactoringCrawler [28]. RefDiff supports the detection of only 12 refactoring types, and cannot be extended to support the detection of sub-method level refactoring types, because it represents the bodies of methods as bags of tokens, and thus the structure of method bodies is completely lost after tokenization.

5.1 Extending GumTreeDiff to Report Refactorings

To build a ground truth for the new submethod-level refactoring types supported by RefactoringMiner 2.0, we relied on GumTreeDiff [37], which is a generic Abstract Syntax Tree (AST) differencing tool that can consume raw source code (i.e., does not require compiled source code) and return changes at the finest level of granularity (i.e., AST leaf nodes). Given the ASTs of two Java compilation units, GumTreeDiff computes the shortest possible script of edit operations to convert one tree to the other. GumTreeDiff reports four edit operations in the computed scripts, namely *Update Value*, *Add*, *Delete*, and *Move* AST node. We gave as input to GumTreeDiff all pairs of Java compilation units having an identical file path in the parent and child commit, and collected all *Update Value* operations in the computed edit scripts. It should be noted that GumTreeDiff does not

TABLE 7
GumTreeDiff refactoring detection rules

Refactoring Type	Rule
Given an *Update Value* operation on a pair of mapped `SimpleName` AST nodes (`src`, `dst`) from the parent and child commit, respectively	
Rename Method	The parent node of `src` is a `MethodDeclaration`
Rename Variable (1)	The parent node of `src` is a `VariableDeclarationFragment`, the grandparent node of `src` is a `VariableDeclarationStatement` or `VariableDeclarationExpression`, and there exist at least two *Update Value* operations involving `src` and `dst` nodes in the compilation unit [1]
Rename Variable (2)	The parent node of `src` is a `SingleVariableDeclaration`, the grandparent node of `src` is an `EnhancedForStatement` or a `CatchClause`
Rename Parameter	The parent node of `src` is a `SingleVariableDeclaration`, and the grandparent node of `src` is a `MethodDeclaration`
Rename Field	The parent node of `src` is a `VariableDeclarationFragment`, the grandparent node of `src` is a `FieldDeclaration`, and there exist at least two *Update Value* operations involving `src` and `dst` nodes in the compilation unit [1]
Given an *Update Value* operation on a pair of mapped `Type` AST nodes (`src`, `dst`) from the parent and child commit, respectively	
Change Return Type	The parent node of `src` is a `MethodDeclaration`
Change Variable Type (1)	The parent node of `src` is a `VariableDeclarationStatement` or `VariableDeclarationExpression`
Change Variable Type (2)	The parent node of `src` is a `SingleVariableDeclaration`, the grandparent node of `src` is an `EnhancedForStatement` or a `CatchClause`
Change Parameter Type	The parent node of `src` is a `SingleVariableDeclaration`, and the grandparent node of `src` is a `MethodDeclaration`
Change Field Type	The parent node of `src` is a `FieldDeclaration`

[1] *to ensure that at least one reference of the variable/field declaration is updated in the same way.*

accept as input two sets of Java compilation units corresponding to the added, deleted, and changed files in a commit, and thus it is impossible to infer refactoring operations involving the move of program elements between different files or packages. The rules we used to infer rename and type-change related refactoring operations are shown in Table 7 and their implementation is publicly available [75].

5.2 Oracle Extension

The original oracle [1] was constructed from a publicly available dataset of refactoring instances [2], comprising 536 commits from 185 open-source GitHub-hosted projects monitored over a period of two months (between June $8th$ and August $7th$, 2015). This dataset is highly reliable, since all instances went through rigorous manual validation by multiple authors and in several cases were confirmed by the developers who actually performed them. It is one of the most representative datasets to date, since all instances are real refactorings found in 185 Java projects from different domains, they are motivated by a variety of reasons [2], and take place along with other changes/refactorings in the same commit.

Tsantalis *et al*. [1], executed the prior version of RefactoringMiner and RefDiff on all 536 commits of the dataset. For the validation process, they created a web application [35], which listed all refactorings reported by the two tools, along with links to the corresponding GitHub commits. Through this web application, the validators were able to inspect the change diff provided by GitHub, and enter their validation and comments. In total, they manually validated 4,108 unique refactoring instances detected by the two tools, out of which 3,188 were true positives and 920 were false positives. The validation process was labor-intensive and involved 3 validators for a period of 3 months (i.e., 9 person-months).

To extend the oracle, we executed RefactoringMiner 2.0, GumTreeDiff and two newer versions of RefDiff, namely version 1.0 and 2.0, on all 536 commits of the dataset, added all new refactorings reported by the tools in the web application [35], and considered the union of all true positives as the ground truth. We extended the code of all aforementioned tools to report the refactoring instances they detect in the format used by RefactoringMiner 2.0, and made the extended tools available in a publicly accessible repository [75] to enable the replication of our experiments. To reduce the effort of the validation process, we also extended the web application to provide direct links to the refactored code elements (i.e., the exact line of code in which the refactored code element is declared on the left and right hand side of the change diff provided by GitHub). This feature allowed us to locate the refactored code elements instantly, while in the manual process we had to first use the browser's Find feature to locate the Java file containing the refactored code element, then scroll through and expand hidden parts of the change diff, until we locate the refactored code element itself.

The first two authors of the paper validated the new refactoring instances for a period spanning over a year, as new features were added to RefactoringMiner 2.0 and more refactoring types were supported. In total, we validated 5,830 new unique refactoring instances, out of which 4,038 were true positives and 1,792 were false positives. To the best of our knowledge, this is the largest refactoring oracle of validated instances to date, including 7,226 true positives in total, for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools.

5.3 Comparison of Precision/Recall With RefDiff

Table 8 shows a comparison of precision and recall between RefactoringMiner 2.0 and three different versions of RefDiff for the commonly detected refactoring types. We can observe that our tool has better precision than all versions of RefDiff for all refactoring types. The precision of RefactoringMiner 2.0 is ranging between 98.2 and 100 percent with an average of 99.7 percent. It has also better recall for all refactoring types, except for Extract & Move Method, for which RefDiff 0.1.1 applies a very lenient similarity threshold, as can be inferred from the low precision and high recall scores it achieves. It should be noted that RefDiff 2.0 discontinued the support of all field-related refactoring types, and thus we were not able to compute its precision and recall for three refactoring types (i.e., the rows in Table 8 with N/A).

Compared to its predecessor, RefactoringMiner 2.0 achieves a much higher recall for all refactoring types. This improvement can be mainly attributed to the extensions made to the tool, such as the implementation of new replacement types (Section 4.3) and heuristics (Section 3.3) to match statements, and the detection of nested refactorings operations (Section 4.4). The new features improve the

TABLE 8
Precision and recall per refactoring type

		RefactoringMiner 2.0		RefactoringMiner 1.0		RefDiff 0.1.1		RefDiff 1.0		RefDiff 2.0	
Refactoring Type	#TP	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Inline Method	107	100	95.3	97.8	85.0	81.0	79.4	97.1	61.7	88.8	66.4
Extract Method	957	99.8	95.8	97.6	75.5	94.6	79.3	98.7	39.8	98.5	61.4
Move Field	249	98.4	96.0	84.1	93.6	32.3	42.6	55.5	44.6	N/A	
Move Class	1091	100	99.3	99.5	93.4	99.3	89.9	99.6	95.8	99.8	97.3
Extract Interface	21	100	100	95.0	90.5	80.0	57.1	61.1	52.4	87.5	100
Push Down Method	43	100	97.7	100	74.4	100	44.2	100	44.2	87.8	83.7
Push Down Field	33	100	100	100	78.8	100	90.9	100	78.8	N/A	
Pull Up Method	291	100	97.9	100	93.5	97.6	28.2	97.1	22.7	96.4	92.1
Pull Up Field	129	100	99.2	100	98.4	100	24.0	100	17.8	N/A	
Move Method	283	99.6	92.9	96.2	79.9	28.6	91.5	58.7	90.8	76.2	80.2
Rename Method	371	98.2	87.9	94.7	67.4	85.8	76.5	94.4	59.6	90.9	62.3
Extract Superclass	71	100	100	100	100	100	18.3	89.3	70.4	98.1	71.8
Rename Class	56	100	89.3	97.4	67.9	92.3	85.7	97.7	76.8	96.1	87.5
Extract & Move Method	166	100	54.2	73.3	19.9	65.6	75.9	91.5	25.9	63.7	39.2
Move & Rename Class	41	100	90.2	70.0	51.2	74.2	56.1	92.0	56.1	80.0	68.3
Move & Inline Method	19	100	73.7	N/A		92.9	68.4	100	15.8	100	57.9
Average	3938	99.7	94.2	96.5	81.3	72.9	73.1	88.3	60.7	93.8	76.9

overall quality of the obtained statement mappings, which in turn boosts the recall of the tool for refactoring types relying on the number of matched/unmatched statements.

As discussed in [1], the Achilles' heel of RefDiff 0.1.1, in terms of precision, is the detection of Move Method and Move Field refactorings (28.6 and 32.3 percent, respectively). We found two recurring scenarios causing such false positives for RefDiff 0.1.1. In the first scenario, RefDiff 0.1.1 misses the detection of a class move to another package, and consequently reports the methods and fields of that class as being moved from the original class, which is assumed to be deleted, to another class, which is assumed to be newly added. In the second scenario, a subclass extending/implementing a given superclass/interface is deleted, and a new subclass is added, which overrides the superclass/interface methods in a similar way. RefDiff 0.1.1 reports these methods as being moved from the deleted to the added subclass. These limitations have been addressed in the next versions of RefDiff, as can be inferred from the increase of recall for Move Class refactoring, which contributed to the increase of precision for Move Method and Move Field refactorings.

A major change between RefDiff 0.1.1 and the next versions, affecting its precision and recall, is related to the similarity thresholds used in the rules that determine the relationships between the program elements in the child and parent commit. RefDiff 0.1.1 uses thresholds that were calibrated based on a randomly selected set of ten commits from ten different projects, drawn from a publicly available dataset of refactoring instances [2], which were confirmed by the developers who actually performed them. On the other hand, the next versions of RefDiff use a single fixed threshold for all relationships, which is equal to 0.5. Since the newly adopted similarity threshold is more conservative than the previously used thresholds (ranging between 0.1 and 0.3), we can observe an increase of precision along with a significant decrease of recall for the refactoring types relying heavily on code similarity, such as Extract Method, Inline Method and Rename Method. This result proves our argument about the inherent limitation of approaches relying on code similarity thresholds to detect refactorings. Even small variations in threshold values can lead to large differences in precision and recall [54].

RefactoringMiner and RefDiff follow contrary approaches in the way they handle source code structure to match code elements. RefactoringMiner relies heavily on the structure of source code statements, while RefDiff ignores it completely by treating code fragments as bags of tokens. Ignoring the structure of source code statements makes the detection approach robust to major structural changes (e.g., merging/splitting of conditionals, as in the case found in project jetty[2]), as long as the variable and method identifiers used in the restructured statements remain the same. On the other hand, treating source code as bags of tokens makes the detection approach unable to deal with changes in the tokens caused by the refactoring itself (e.g., parameterization of expressions in Extract Method refactoring), or other overlapping refactorings (e.g., local variable renames inside the body of a refactored method). RefactoringMiner deals robustly with such token changes by applying statement pre-processing techniques, such as argumentization, and allowing syntax-aware replacements of AST nodes within matched statements, which are further utilized to infer sub-method-level refactorings. Further research on hybrid methods that combine the advantages of RefactoringMiner and RefDiff seems to have great potential.

5.4 Comparison of Precision/Recall With GumTreeDiff

Table 9 shows a comparison of precision and recall between RefactoringMiner 2.0 and GumTreeDiff 2.1.2 for the commonly detected refactoring types. We can observe that our

2. https://github.com/eclipse/jetty.project/commit/1f3be6#diff-ff02a462f6cc50644669e515c691229dR580

TABLE 9
Precision and recall per refactoring type

		RefactoringMiner 2.0		GumTreeDiff 2.1.2	
Refactoring Type	#TP	Precision	Recall	Precision	Recall
Rename Method	371	98.2	87.9	28.2	74.9
Rename Variable	273	98.8	93.8	65.1	56.8
Rename Parameter	497	99.3	91.3	64.4	66.6
Rename Field	133	99.1	85.7	78.3	70.7
Change Return Type	400	99.5	96.5	65.3	52.3
Change Variable Type	691	99.7	93.9	82.4	60.9
Change Parameter Type	632	99.8	94.5	80.1	63.8
Change Field Type	222	99.5	95.5	75.0	62.2
Average	3219	99.4	93.0	60.1	63.0

tool has better precision and recall than GumTreeDiff for all refactoring types commonly detected by the two tools.

There are three main factors that contribute negatively to the precision and recall of GumTreeDiff, with a notable lowest precision of 28.2 percent for Rename Method refactoring. 1) GumTreeDiff is a language-independent tool, and thus it may match AST nodes that are semantically incompatible. For example, we found many cases where the formal parameter of an enhanced-for statement is matched with a parameter of a method declaration, because they both have the same AST node type (i.e., `SingleVariableDeclaration`).

2) The addition of new methods or the reordering of methods in a commit, may result in incorrect AST node matches, because the applied tree differencing algorithm tends to match the method declarations in the order they appear inside their parent type declarations. For example, in project aws-sdk-java[3], the position of 130 pairs of getter and setter methods has been swapped, and GumTreeDiff matched all getters with their corresponding setters and vice versa, resulting in 260 Rename Method false positives within a single commit.

3) The applied tree differencing algorithm is not refactoring-aware. For example, when a relatively large portion of a method is extracted to a new method, GumTreeDiff tends to match the original method in the parent commit with the extracted method in the child commit, resulting in Rename Method false positives.

On the other hand, our tool matches method declarations based on their signatures (regardless of their relative location in their parent type declarations) in the first phase, and then matches the remaining unmatched method declarations with potential signature changes based on their source code contents in the second phase (Section 4.2). In addition, the rule we defined for detecting methods with changes in their signature (Table 4), takes into consideration the presence of overlapping Extract Method or Inline Method refactorings, which may change significantly the source code contents of a method, making more challenging the matching of renamed methods. Finally, since our tool is specifically designed to analyze Java code, it is semantic-aware and does not allow to match AST nodes appearing in semantically incompatible contexts.

3. https://github.com/aws/aws-sdk-java/commit/4baf0

TABLE 10
Precision for refactoring types detected by RefactoringMiner 2.0

Refactoring Type	TP	Precision	Refactoring Type	TP	Precision
Replace Field w/ Field	1	100	Merge Variable	4	100
Extract Subclass	5	100	Merge Parameter	26	100
Extract Class	53	100	Merge Field	5	100
Change Package	27	100	Split Variable	1	100
Extract Variable	112	100	Split Parameter	6	100
Extract Field	5	100	Split Field	2	100
Inline Variable	31	100	Move & Rename Field	6	100
Parameterize Variable	38	100	Replace Variable w/ Field	16	100

5.5 Precision for Refactoring Types Supported Only by RefactoringMiner 2.0

Table 10 shows the number of true positives and precision for the refactoring types detected only by RefactoringMiner 2.0. In our manual inspection of the detected instances, we did not find any false positives, and thus the precision is 100 percent for all refactoring types, similar to the general precision trend we observed for all refactoring types. Obviously, it is quite possible that our tool missed some true instances, but we did not find any tool operating at commit level that can detect these refactoring types to build a more reliable ground truth.

5.6 Comparison of Execution Time

Fig. 5 shows the distribution of the execution time of RefactoringMiner 2.0 and 1.0 and RefDiff 2.0, 1.0, and 0.1.1 for all 536 commits in our oracle (the y-axis has a logarithmic scale). We executed separately each tool on the same machine with the following specifications: Intel Core i7-3770 CPU @ 3.40GHz, 16 GB DDR3 memory, 256GB SSD, Windows 10 OS, and Java 11.04 x64. For each tool, we recorded the time taken for parsing the source code of the examined and its parent commit, and the time taken to detect refactorings using the `System.nanoTime` Java method. To make a fair comparison, we excluded the time needed to checkout commits, or access the *.git* folder of the repository, because these two different approaches to obtain the added, deleted, and changed files between the parent and child commits have a vast computational cost difference, as we will discuss later.

On median, RefactoringMiner 2.0 is 2.6× faster than RefDiff 2.0 and 2.3× faster than its predecessor, while it is 31× and 33× faster than RefDiff 0.1.1 and 1.0, respectively. On average, RefactoringMiner 2.0 is 1.2× faster than RefDiff 2.0 and 5.8× faster than its predecessor, while it is 11× and 11.5× faster than RefDiff 0.1.1 and 1.0, respectively. As it can be observed from the first violin plot shown in Fig. 5, RefactoringMiner 2.0 is able to process each commit in our oracle in less than 5 seconds, except for one commit, which takes 13 seconds and corresponds to the Max value of the distribution. To assess if there is a statistical difference among these distributions, we perform the Kruskal-Wallis test. We reject the null hypothesis that the medians of all distributions are equal (p-value = 1.7×10^{-3}). We also perform the pair-wise post-hoc Dunn's test to compare the distributions with visibly close medians. We reject the null hypothesis that the distribution of RefactoringMiner 2.0 has an equal median with that of RefactoringMiner 1.0 (p-value = 3.6×10^{-28}) and RefDiff 2.0 (p-value = 4.1×10^{-52}).

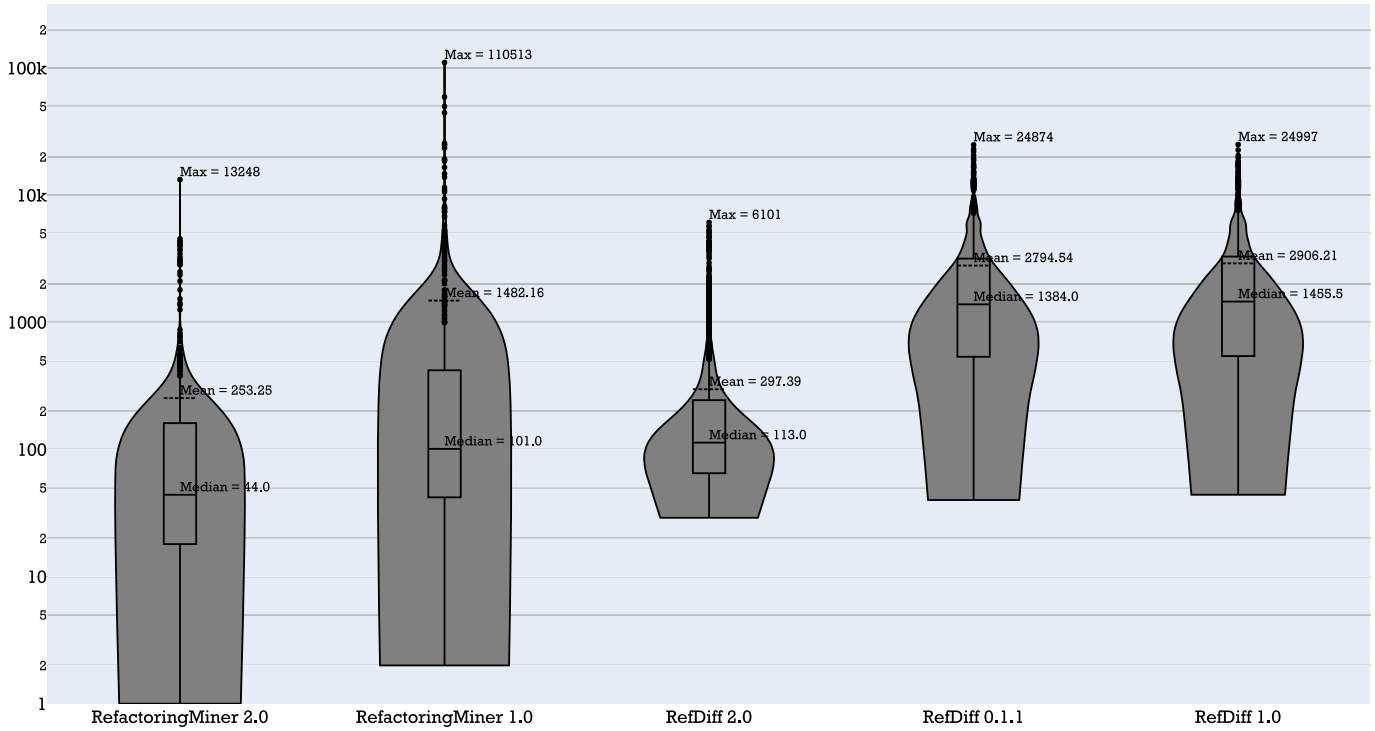

Fig. 5. Execution time per commit in milliseconds.

The main performance bottleneck in the algorithm of RefactoringMiner is the matching of large statements, which include anonymous class declarations or lambda expressions. Performing all possible combinations of AST node replacements to match large statements (Algorithm 2) can lead to a combinatorial explosion, when the number of uncommon AST nodes between the statements is large. To overcome this problem, we match separately the statements inside the methods of anonymous class declarations and the bodies of lambda expressions. If the number of matching statements is larger than the number of unmatched statements, then we consider the parent statements as matching without performing AST node replacements. This change explains to a large extent the improvement in the execution time of RefactoringMiner 2.0 over its predecessor.

RefDiff 0.1.1 and 1.0 require to *checkout* the parent and child commits in a repository to detect the refactoring operations performed in the child commit. The checkout command updates the files in the working tree to match the version in the index of the specified tree (i.e., commit). By performing checkout, RefDiff has access to all repository files in the version corresponding to a given commit. Then, RefDiff extracts the source folders of the repository to setup the Eclipse JDT ASTParser in a way that enables the resolution of partial binding information, which is used to infer method call and type hierarchy relationships between code elements. Although the refactoring detection is performed only on the added, deleted, and changed files between the parent and child commits, the binding information is resolved using the entire repository. The checkout command is a hard disk write-operation, and thus it has a vast execution time overhead, especially for large repositories. To give an idea about the checkout overhead, it takes 4 hours to process all 536 commits by checking out all parent and child commits (i.e., 2×536 commits). For very large repositories, checking out a single commit can take several minutes. The difference in the execution time between RefDiff 0.1.1 and 1.0, and the other three tools, as shown in Fig. 5, is mainly due to the fact that prior RefDiff versions were setting up the Eclipse JDT ASTParser to perform binding resolution, and thus the ASTParser was analyzing all source folders of the repository introducing an additional execution time overhead. This overhead along with the checkout overhead makes RefDiff 0.1.1 and 1.0 unsuitable for performing *live* refactoring detection, and very slow for mining refactoring operations from the entire commit history of a project.

On the other hand, RefactoringMiner 2.0 uses the JGit API to obtain the contents of the added, deleted, and changed files between the parent and child commits directly from the *.git* folder of the repository. This is a hard disk read-operation that takes significantly less time than executing the checkout command two times for each analyzed commit, and allows to mine refactoring operations from the entire commit history of a project with the least overhead (i.e., one hard disk write-operation to clone the repository locally, and one hard disk read-operation to load the *.git* folder information in memory). In addition, RefactoringMiner 2.0 can fetch the contents of the added, deleted, and changed files directly from GitHub using the GitHub API. This feature enables a series of applications that require *live* refactoring information for a given commit, without having to clone locally the repository under analysis. To give an idea about the execution time gain over the checkout approach, it takes less than 20 minutes to process all 536 commits using the JGit API with locally cloned repositories, and 40 minutes using the GitHub API to download the files.

RefDiff 2.0 adopts a similar to RefactoringMiner 2.0 strategy for obtaining and parsing the contents of the added,

deleted, and changed files, which explains the significant reduction of execution time over its predecessors. Since both tools use the same strategy for obtaining and parsing the files, the difference in their execution time can be attributed solely to the algorithms used for detecting refactorings. RefactoringMiner 2.0 has a twice faster detection algorithm, despite the fact that it supports a much larger number of refactoring types (40 versus 13).

5.7 Limitations and Threats to Validity

Missing Context. As explained in Section 4.1, RefactoringMiner 2.0 analyzes only the added, deleted, and changed files between two revisions. However, the missing context (i.e., the unchanged files) can make the tool report an incorrect refactoring type for certain operations. For example, if a method or field is pulled multiple levels up to the inheritance hierarchy and some classes between the source and destination are unchanged, then RefactoringMiner 2.0 will report it as a move, because it cannot detect the inheritance relationship between the source and destination classes due to the missing context. In our oracle, this scenario occurred only once in project cascading[4], where four methods were pulled three levels up `TezNodeStats` → `BaseHadoopNodeStats` → `FlowNodeStats` → `CascadingStats`, but class `FlowNodeStats` remained unchanged in the commit. Given that processing all repository source code files has a tremendous performance overhead, as shown in the experiments with prior versions of RefDiff (Section 5.6), and the only advantage is labelling properly certain refactoring types in rare scenarios, we conclude that missing context is not a major weakness of our tool.

Language Specificity. RefactoringMiner 2.0 currently supports only Java programs. Extending the tool to support other languages has challenges that go beyond a simple engineering task. First, there are language-specific refactoring types, which are not applicable in other programming paradigms, or even languages from the same paradigm. For example, Go is an object-oriented language that uses *struct embedding* (i.e., type composition) instead of inheritance, making all inheritance-related refactorings inapplicable. Second, the type system used by a language affects the source code matching process. For example, in dynamically typed languages, function signatures consist only of the function name and number of arguments, thus making signature-based matching more ambiguous than statically typed languages. The matching of variable declaration statements is also more ambiguous in dynamically typed languages, because we can rely only on variable identifiers and optional initializer expressions to match variable declarations, as variable types are not available in the source code.

External Validity (Generalizability to Unsupported Refactoring Types). In this work, we present and evaluate the detection rules for 40 refactoring types. Fowler's book [30] covers around seventy different types, and several new types have been added to the book's companion website over time. Despite not supporting all refactoring types, we have shown that our approach based on statement mapping information is reliable and capable of achieving very high precision and recall for both high-level and low-level refactoring types. Thus, we conclude that any refactoring type that can be detected based on statement mapping information will also have high precision and recall.

Internal Validity (Experimenter Bias). Although we did our best effort to reduce bias in the construction of our oracle by incorporating the input of six different tools and manual validations by multiple authors, we cannot claim the oracle is completely unbiased, as all validators are co-authors of this work and our previous one [1]. Overall, around 72 percent of the true refactoring instances in our oracle are detected by two or more tools. More specifically, 2,117 are detected by two tools, 555 by three tools, 689 by four tools, 1,711 by five tools, and 144 by six tools. In addition, two up to four validators inspected around eight percent of the refactoring instances (671 instances were validated by two, 144 by three, and 12 by four different validators), which were more challenging to analyze. In general, the vast majority of the detected instances were straightforward to inspect and assess their validity.

6 Conclusion

In this work, we presented the newer version of our refactoring mining tool. RefactoringMiner 2.0 has some unique features that distinguish it from other competitive tools: (1) it does not rely on code similarity thresholds, (2) it supports low-level refactorings that take place within the body of methods, (3) it can detect nested refactoring operations within a single commit. To evaluate our tool, we created one of the most accurate, complete, and representative refactoring oracles to date, including 7,226 true instances for 40 different refactoring types detected by one (minimum) up to six (maximum) different tools, and validated by one up to four refactoring experts. Our evaluation showed that our approach achieves the highest average precision (99.6 percent) and recall (94 percent) among all competitive tools, and on median is 2.6 times faster than the second faster competitive tool (on median, it takes 44 ms to process a commit in our oracle).

Implications. The high accuracy and fast execution time of RefactoringMiner 2.0, along with its ability to operate on the commits of a locally cloned repository, or fetch the contents of the added/deleted/changed files in a commit directly from GitHub, enable novel applications:

1) We can provide *live* refactoring information to assist the code review process. As a matter of fact, we created an extension for the Chrome browser [76] that enhances the GitHub commit diff webpage with overlaid refactoring information.
2) Empirical researchers can create refactoring datasets fast and with high precision from the commit history of projects, and study various software evolution phenomena at the finest level of granularity.
3) Several code evolution analysis techniques that are susceptible to refactoring *noise* can become *refactoring-aware* and improve their accuracy.
4) Refactoring operations can be automatically documented at commit-time to provide a more detailed description of the applied changes in the commit message, and help in the untangling of behavior-preserving changes from behavior-altering changes.

4. https://github.com/cwensel/cascading/commit/f9d31

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their insightful and constructive feedback for improving the paper. The authors would like to thank Dr. Davood Mazinanian for the features he implemented in the web application used for validating the refactoring instances in our oracle. This research was partially supported by NSERC grant RGPIN-2018-05095 and NSF grant CCF-1553741.

REFERENCES

[1] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 483–494.

[2] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 858–870.

[3] S. Fakhoury, D. Roy, S. A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *Proc. 27th Int. Conf. Program Comprehension*, 2019, pp. 2–12.

[4] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "An empirical investigation of how and why developers rename identifiers," in *Proc. 2nd Int. Workshop Refactoring*, 2018, pp. 26–33.

[5] B. Lin, C. Nagy, G. Bavota, and M. Lanza, "On the impact of refactoring operations on code naturalness," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng.*, Feb 2019, pp. 594–598.

[6] D. Cedrim *et al.*, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Proc. 11th Joint Meet. Foundations Softw. Eng.*, 2017, pp. 465–475.

[7] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, "Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?" in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2019, pp. 186–190.

[8] W. Lucas, R. Bonifácio, E. D. Canedo, D. Marcílio, and F. Lima, "Does the introduction of lambda expressions improve the comprehension of Java programs?" in *Proc. 33th Brazilian Symp. Softw. Eng.*, 2019, pp. 187–196.

[9] M. Mahmoudi, S. Nadi, and N. Tsantalis, "Are refactorings to blame? an empirical study of refactorings in merge conflicts," in *Proc. 26th Int. Conf. Softw. Anal., Evol. Reeng.*, 2019, pp. 151–162.

[10] Palantir Technologies. Code review best practices. 2018. [Online]. Available: https://medium.com/palantir/code-review-best-practices-19e02780015f

[11] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.

[12] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 81–90.

[13] C. Williams and J. Spacco, "SZZ revisited: Verifying when changes induce fixes," in *Proc. Workshop Defects Large Softw. Syst.*, 2008, pp. 32–36.

[14] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.

[15] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: Cost negotiation and community values in three software ecosystems," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 109–120.

[16] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 357–366.

[17] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 3, pp. 321–335, May 2008.

[18] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "IntelliMerge: A refactoring-aware software merging technique," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 170:1–170:28, Oct. 2019.

[19] E. C. Neto, D. A. da Costa, and U. Kulesza, "Revisiting and improving SZZ implementations," in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2019, pp. 1–12.

[20] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 70–79.

[21] M. Mahmoudi and S. Nadi, "The android update problem: An empirical study," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 220–230.

[22] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "APIDiff: Detecting API breaking changes," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reengineering*, 2018, pp. 507–511.

[23] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 233–244.

[24] Z. Chen, H. Guo, and M. Song, "Improving regression test efficiency with an awareness of refactoring changes," *Inf. Softw. Technol.*, vol. 103, pp. 174–187, 2018.

[25] E. L. G. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 751–754.

[26] X. Ge, S. Sarkar, and E. Murphy-Hill, "Towards refactoring-aware code review," in *Proc. 7th Int. Workshop Cooperative Hum. Aspects Softw. Eng.*, 2014, pp. 99–102.

[27] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill, "Refactoring-aware code review," in *Proc. IEEE Symp. Visual Languages and Human-Centric Comput.*, Oct 2017, pp. 71–79.

[28] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 269–279.

[29] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

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

[31] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *Proc. 18th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2010, pp. 371–372.

[32] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proc. 20th Eur. Conf. Object-Oriented Program.*, 2006, pp. 404–428.

[33] Z. Xing and E. Stroulia, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Proc. Companion 30th Int. Conf. Softw. Eng.*, 2008, pp. 951–952.

[34] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. 27th Eur. Conf. Object-Oriented Program.*, 2013, pp. 552–576.

[35] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and A. Ketkar. Refactoring oracle. 2019 [Online]. Available: http://refactoring.encs.concordia.ca/oracle/

[36] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2020.2968072.

[37] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 313–324.

[38] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proc. 15th ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang. Appl.*, 2000, pp. 166–177.

[39] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *Proc. 7th Int. Workshop Principles Softw. Evol.*, 2004, pp. 31–40.

[40] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[41] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 231–240.

[42] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[43] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Proc. 13th Work. Conf. Reverse Eng.*, 2006, pp. 263–274.

[44] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2005, pp. 54–65.

[45] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[46] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 211–221.
[47] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 222–232.
[48] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1095–1105.
[49] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
[50] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, 2012, special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.
[51] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *Proc. IEEE Conf. Softw. Maintenance Reengineering Reverse Eng.*, 2014, pp. 254–263.
[52] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proc. 6th Int. Workshop Emerg. Trends Softw. Metrics*, 2015, pp. 44–53.
[53] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016.
[54] D. Dig, "Automated upgrading of component-based applications," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2007.
[55] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "Satt: Tailoring code metric thresholds for different software architectures," in *Proc. 16th Int. Work. Conf. Source Code Anal. Manipulation*, 2016, pp. 41–50.
[56] M. Tufano *et al.*, "There and back again: Can you compile that snapshot?" *J. Softw.: Evol. Process*, vol. 29, no. 4, 2017, Art. no. e1838.
[57] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: An approach for the automated generation of release notes," *IEEE Trans. Softw. Eng.*, vol. 43, no. 2, pp. 106–127, Feb. 2017.
[58] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *J. Syst. Softw.*, vol. 86, no. 4, pp. 1006–1022, Apr. 2013.
[59] I. Kádár, P. Hegedűs, R. Ferenc, and T. Gyimóthy, "A manually validated code refactoring dataset and its assessment regarding software maintainability," in *Proc. 12th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 2016, pp. 10:1–10:4.
[60] L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Towards a benchmark for evaluating design pattern miner tools," in *Proc. 12th Eur. Conf. Softw. Maintenance Reengineering*, 2008, pp. 143–152.
[61] G. Kniesel and A. Binun, "Standing on the shoulders of giants - a data fusion approach to design pattern detection," in *Proc. 17th Int. Conf. Program Comprehension*, 2009, pp. 208–217.
[62] F. A. Fontana, A. Caracciolo, and M. Zanoni, "DPB: A benchmark for design pattern detection tools," in *Proc. 16th Eur. Conf. Softw. Maintenance Reengineering*, 2012, pp. 235–244.
[63] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 456–460.
[64] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proc. 26th Eur. Conf. Object-Oriented Program.*, 2012, pp. 79–103.
[65] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.
[66] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
[67] G. Kondrak, "N-gram similarity and distance," in *Proc. 12th Int. Conf. String Process. Inf. Retrieval*, 2005, pp. 115–126.
[68] Q. D. Soetens, J. Pérez, S. Demeyer, and A. Zaidman, "Circumventing refactoring masking using fine-grained change recording," in *Proc. 14th Int. Workshop Principles Softw. Evol.*, 2015, pp. 9–18.
[69] M. Fowler. Fluent interface. 2005. [Online]. Available: https://martinfowler.com/bliki/FluentInterface.html
[70] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer, "Comparison of similarity metrics for refactoring detection," in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 53–62.
[71] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 17, no. 5, pp. 309–332, Sep. 2005.
[72] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proc. 2nd Int. Conf. Extreme Program. Flexible Processes Softw. Eng.*, 2001, pp. 92–95.
[73] A. Sherman. Always squash and rebase your git commits. 2017. [Online]. Available: https://blog.carbonfive.com/2017/08/28/always-squash-and-rebase-your-gi% t-commits/
[74] S. Carletti. Two years of squash merge. 2019. [Online]. Available: https://blog.dnsimple.com/2019/01/two-years-of-squash-merge/
[75] A. Ketkar and N. Tsantalis. Tools used for the evaluation of refactoringminer 2.0. 2019. [Online]. Available: https://github.com/ameyaKetkar/RMinerEvaluationTools
[76] H. Mansour and N. Tsantalis. Refactoring aware commit review chrome extension. 2019. [Online]. Available: https://chrome.google.com/webstore/detail/refactoring-aware-commit/lnlo% iaibmonmmpnfibfjjlfcddoppmgd

Nikolaos Tsantalis (Senior Member, IEEE) is an associate professor with the department of Computer Science and Software Engineering at Concordia University, Montreal, Canada, and holds a Concordia University research chair in Web Software Technologies. His research interests include software maintenance, software evolution, empirical software engineering, refactoring recommendation systems, refactoring mining, and software quality assurance. He has been awarded with two Most Influential Paper awards at SANER 2018 and SANER 2019, and two ACM SIGSOFT Distinguished Paper awards at FSE 2016 and ICSE 2017. He served as a program co-chair for various tracks in SANER, SCAM and ICPC conferences, and serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, MSR, SANER, ICPC, and SCAM. He is a member of the *IEEE Transactions on Software Engineering* review board. Finally, he is a senior member the ACM, and holds a license from the Association of Professional Engineers of Ontario.

Ameya Ketkar is currently working toward the PhD degree at Oregon State University. His research interests include program transformations that improve software quality and empirical studies that investigate how programming language features are used in practice. He has received a Distinguished Artifact Award at OOPSLA'17 and was a runner-up at Microsoft Student Research Competition at FSE'18. He also served as a member of the program committee of the ICPC Tool Demonstration track, in 2020.

Danny Dig (Member, IEEE) is an associate professor with the department of Computer Science at University of Colorado Boulder, and an adjunct professor with the University of Illinois and Oregon State University. He enjoys doing research in Software Engineering, with a focus on interactive program transformations that improve programmer productivity and software quality. He successfully pioneered interactive program transformations by opening the field of refactoring in cutting-edge domains including mobile, concurrency and parallelism, component-based, testing, and end-user programming. He coauthored more than 50 journal and conference papers that appeared in top places in SE/PL. His research was recognized with eight best paper awards at the flagship and top conferences in SE (FSE'17, ICSME'17, FSE'16, ICSE'14, ISSTA'13, ICST'13, ICSME'15), four award runner-ups, and one most influential paper award (N-10 years) at ICSME'15.