IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 4, 2021, accepted May 10, 2021, date of publication June 4, 2021, date of current version June 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3086689

RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment

IMAN HEMATI MOGHADAM[1], MEL Ó CINNÉIDE[2], FAEZEH ZAREPOUR[3], AND MOHAMAD AREF JAHANMIR[1]

[1]Department of Computer Engineering, Vali-e-Asr University of Rafsanjan, Rafsanjan 7718897111, Iran
[2]School of Computer Science, National University of Ireland, Dublin, D04 V1W8 Ireland
[3]Department of Computer Engineering, Allameh Jafari Institute of Rafsanjan, Rafsanjan 77181, Iran

Corresponding author: Iman Hemati Moghadam (i.hemati@vru.ac.ir)

This work was supported by Science Foundation Ireland grant 13/RC/2094_2 to Lero - the Science Foundation Ireland Research Centre for Software.

ABSTRACT Refactoring is performed to improve software quality while leaving the behaviour of the software unchanged. Identifying refactorings applied to a software system is an important activity that leads to a better understanding of the evolution of the software system, and several techniques have been proposed and implemented to address this issue. The vast majority of existing refactoring detection techniques are language-specific, including the accepted state of the art, RMiner, which is exclusively Java-based. Although impressive performance has been achieved to date, there is scope for improvement in refactoring detection and such improvement would enhance both refactoring research and practice. In this paper, we propose a novel, language-neutral technique to identify refactorings in commit histories. Our approach is motivated by a desire to explore the use of string alignment algorithms in refactoring detection, and to determine if such approaches are competitive with the state of the art. The proposed approach has been implemented in a tool called RefDetect, evaluated, and compared with the current state-of-the-art refactoring detection tool: RMiner. In experiments we applied RefDetect to 514 commits of 185 Java applications containing 5,058 true refactoring instances, achieving an f-score slightly better than that achieved by RMiner (87.3% vs. 86%). RefDetect clearly outperformed RMiner in method and class based refactorings, achieving f-scores respectively of 87.7% vs. 81.7% for method-level refactorings and 92.1% vs. 86.9% for class-level refactorings. To demonstrate the language-independence of RefDetect, we conducted a further study with four C++ applications, achieving high values for both precision (96.1%) and recall (94.1%). The achieved results indicate that RefDetect performs better than the current state of the art in refactoring detection and is demonstrably capable of handling different programming languages.

INDEX TERMS Refactoring, refactoring detection, alignment algorithm, empirical studies, Java, C++.

I. INTRODUCTION

Refactoring is a key practice in contemporary software development. In Agile development, where little upfront design is performed, it is the practice that enables the design of the software to evolve. It also plays a central role in Test-Driven Development and is regarded as an essential practice in keeping the codebase ''clean'' and amenable to further development. It has also attracted considerable interest from researcher, with a recent survey paper [1] finding over 3,000 papers on refactoring topics including refactoring at different levels (from architecture to code), applied in many domains e.g. cloud computing, mobile development, web development, and applied for many purposes including to improve software design (the most common goal of refactoring), and also to improve software performance, software security, and most recently to reduce the energy consumption of software.

Given this interest in refactoring as a research topic and its practical importance in software development, it is not surprising that identification of refactorings has been an active research topic for many years [2]–[12]. For practitioners, understanding the nature and extent of the refactorings applied to a system helps in understanding how the system has evolved and how it has been maintained. For refactoring researchers, the starting point for their investigations frequently involves the identification of refactorings that have been applied as the project evolved. According to a

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

recent survey spanning 86 articles [13], the most investigated applications for detecting changes that occurred between two program versions are in order: *understanding system evolution*, *reperforming changes in different contexts*, *change impact analysis*, *correlating changes to other metrics*, and *predicting/suggesting future changes*. On the other hand, *detecting patterns of change* and *reperforming changes in different contexts* are mentioned as the most desirable future applications.

Many attempts have been made to automate the process of refactoring detection [4]–[12], with the current state of the art being represented by the RMiner tool, the work of Tsantalis *et al.* [2], [3]. In experiments, RMiner proved itself capable of detecting 40 refactorings types with a precision and recall of 99.6% and 94% respectively. While these results are impressive, and RMiner has proved to be a very valuable tool in refactoring research, it does not mean that other approaches to refactoring detection should not be explored.

In this paper, we present a novel approach to refactoring detection based on *string alignment*, which we embody in our tool, RefDetect. While RMiner uses the ASTs of two consecutive commits to determine what refactorings have occurred, our approach models the software design embodied in each commit as a string and then employs a string alignment algorithm, FOGSAA [14], as a basis for determining the changes that have occurred and detecting which of them represent refactoring operations. Our approach employs two passes, and uses refactorings detected in the first pass to enhance detection in the second pass. As it is string-based, it depends purely on language-independent entities and relationships and can therefore be easily extended to other programming languages.

In contrast to RMiner, our approach relies on thresholds. However these thresholds are very general and in our experiments we calibrate them only once based on a small number of commits (10) and a number of seed refactorings from edge cases observed in the implementation of RefDetect. These thresholds remain unchanged throughout our experiments involving 514 commits of 185 Java applications, which gives confidence that they do not need to be adjusted on a per-application basis. The two-step approach employed by RefDetect also helps to reduce issues related to similarity thresholds. In the first round, weak thresholds are used mainly to detect rename refactorings as well as new and deleted entities. In the second round, stronger thresholds are used to filter out false positive cases (i.e. improve precision) and also use refactorings detected in the first round to reduce false negatives (i.e. improve recall).

In experiments we applied RefDetect to 514 commits of 185 Java applications containing 5,058 true refactoring instances, achieving an f-score slightly better than that achieved by RMiner (87.3% vs. 86%) on the same dataset. RefDetect clearly outperformed RMiner in method and class based refactorings, achieving f-scores respectively of 87.7% vs. 81.7% for method-level refactorings and 92.1% vs. 86.9% for class-level refactorings. To demonstrate the language-independence of RefDetect, we conducted a further study with four C++ applications, achieving high values for both precision (96.1%) and recall (94.1%). These results indicate that RefDetect performs somewhat better than the current state of the art in refactoring detection, and is demonstrably capable of handling different programming languages.

This paper makes the following contributions:

(1) We present a novel approach based on an alignment algorithm to detect refactorings applied between two program versions. The proposed approach is independent from any programming language, although Java and C++ languages are used to evaluate the approach.

(2) We implement our refactoring detection algorithm as a tool that operates on Java and C++ applications. RefDetect currently supports 27 refactoring types, including composite refactorings.

(3) We compare RefDetect with the current state-of-the-art refactoring detection tool: RMiner [3], where an extensive empirical study on 514 commits from 185 open source Java repositories including 5,058 true refactorings are performed (Section IV-C).

(4) We compare the memory consumption and execution time of our tool with RMiner [3], where the results show that RefDetect is efficient and performs better than RMiner in a number of key ways (Section IV-C4).

(5) We evaluate our tool with four C++ applications including 305 true refactorings where 96.1% precision, and 94.1% recall achieved (Section IV-D).

The rest of the paper is structured as follows. Section II provides an overview and discussion of related works. Our approach to automatic detection of the refactorings that occurred between two program versions is presented in Section III. The correctness and completeness of our approach is evaluated in Section IV, while the limitations of the proposed approach and threats to the validity of our study are discussed in Sections V and VI respectively. Finally, in Section VII, we provide our conclusions and discuss possible related future research.

II. RELATED WORKS

Detecting changes that occurred between two program versions and categorizing them as refactoring instances is investigated in several research works. A large part of the work in this field are based on differencing algorithms to detect if changes applied between two versions of a system can be categorized as refactoring instances. This line of work is related to ours as we use an alignment algorithm as a differencing algorithm to detect changes applied between two versions of a software system.

In an ideal case, using a differencing algorithm the refactorings that take place between two program versions can be detected based only on the original and new versions of the system. However, approaches based on differencing algorithms suffer from two drawbacks: Firstly, they cannot

determine hidden changes,[1] and secondly, they cannot determine the order of applied refactorings. In the following paragraphs we discuss some influential related works in this regard in a chronological order.

The first refactoring detection approach based on a differencing algorithm introduced by Demeyer *et al.* [4]. They use a set of heuristics defined as a combination of low-level source code metrics to identify occurrence of likely refactorings in two successive versions of a program [4]. As an example, the *Extract Superclass* refactoring is detected when the depth of inheritance tree is increased, and the number of methods and fields in a class in the hierarchy structure is decreased while they are increased in the new class added in the hierarchy structure. However, the evaluation results show a low precision for four different types of refactorings detected by this approach. A reason for a low precision is because of partial overlap between the heuristics where some false negative refactorings were reported as false positives for other refactorings [4]. The proposed approach, as confirmed by Demeyer *et al.*, was also vulnerable to renaming and it was also imprecise when many changes are applied on the same piece of code [4].

Weißgerber and Diehl [5] propose a signature-based technique to identify candidate refactorings and use a token-based code clone detection tool (i.e., *CCFinder* [15]) to rank the identified candidates. In the proposed approach, as first step, pairs of code entities (classes, methods, and fields) which have some similarities in their signatures and in the given versions of the program they are removed, added or changed are compared and a candidate refactoring is detected if applied changes satisfy conditions for a refactoring type [5]. To rank the detected refactorings, for each candidate refactoring, the bodies of corresponding entities are compared and, if they are similar, then it is more likely that the changes do not transform the behaviour of the code and the candidate refactoring is more likely to be a correct one. The approach is evaluated on two open-source Java projects and a good level of recall (89.5%) was achieved. However, the evaluation is based on refactorings determined manually from commit logs, and it has been proven that commit messages are not reliable indicators of refactoring activity [12], [16]. It is also worthy of note, as confirmed by Weißgerber and Diehl, that the approach is deficient in finding refactorings when several refactorings are performed on the same entity [5].

In a similar work, Dig *et al.* [6] develop RefactoringCrawler[2] as an Eclipse plug-in which uses a text-based similarity metric (i.e., *Shingles encoding* [17]) to find corresponding entities. However, they use a precise semantic analysis based on *reference graphs* to refine the candidate refactorings. The evaluation results on three open-source Java projects show a high values for both recall (90%) and precision (96%) metrics. However, as with the work of Weißgerber and Diehl [5], the evaluation is based on refactorings determined manually from commit logs. Later, Biegel *et al.* [18] replicated Weißgerber's approach with CCFinder, and two other different similarity metrics: JCCD [19] and Shingles [17], and they report that while the choice of similarity metric has an effect on the ranking of candidate refactorings, the overall results are of comparable quality [18].

Xing and Stroulia [7] develop an Eclipse plug-in named JDEvAn which uses both lexical and structural similarity to automatically recover refactorings which have been occurred between two versions of a program. JDEvAn initially extracts two UML logical design models from the source code of two versions of a Java program, and then uses a differencing algorithm called UMLDiff to detect differences between two models in terms of removal, addition, moving, and renaming of UML entities. The detected differences are then categorized, where possible, as design-level refactoring instances using certain predefined queries [20]. As an example of an implemented query, an *Extract Hierarchy* refactoring is recognized if (1) a new class is added to the program, (2) the new class inherits from at least one class in the program other than Java.lang.object, and (3) the superclass of one or more classes is changed to the new added class [7]. Evaluation results on several case studies prove the quality and usefulness of the JDEvAn in detecting different types of refactorings. However, as confirmed by Xing and Stroulia, there are three situations where UMLDiff cannot detect rename and move refactorings properly: (1) when the renamed or moved entities have few relations with other parts of program, (2) when two unrelated entities have similar names and relations, and finally (3) when other parts of program related to renamed or moved entities are also changed significantly [7]. The latter case was also observed in our earlier work [21] when a hierarchy structure was changed radically but related moved methods and fields could not be detected properly by UMLDiff.

Perte *et al.* [8] develop Ref-Finder,[3] an Eclipse plug-in able to detect 63 refactoring types which is the most comprehensive list of refactoring types to date. Ref-Finder as first step traverses the abstract syntax tree of one version of a program and the next and represents their code entities (e.g. classes, methods and fields), their structural dependencies (e.g. field access and method invocation), and content of the code entities (e.g. method body) as a set of logic predicates. Supported refactorings are also encoded as logic queries, and used to query the set of extracted logic predicates to identify applied refactoring instances [8]. Ref-Finder is the first tool capable of detecting *composite* refactorings where each refactoring consists of a sequence of atomic refactorings. This is done as while encoding refactorings as logic queries, the dependencies between refactorings are included in the definition, and

[1]For example, where a method is moved around various classes before being placed in its final target class, a refactoring detection algorithm based on a differencing algorithm can only detect that the method has been moved from its original class to the target one, and is unable to detect other, hidden *Move Method* refactorings.

[2]http://dig.cs.illinois.edu/tools/RefactoringCrawler

[3]https://sites.google.com/site/reffindertool/

composite refactorings are detected by querying the identified atomic refactorings [8]. For example, an *Extract Superclass* refactoring is detected if a new superclass is created and a number of PullUp Method/Field refactorings are identified that move fields and methods to the newly created class. The accuracy of Ref-Finder is evaluated on three open-source Java programs and a high recall (95%) and an acceptable level of precision (79%) is reported [8]. However, the evaluation was based on refactorings applied in isolation (known as *root canal* refactorings [16]) while in real-world scenarios, refactorings (known as *floss refactorings* [16]) overlap with other changes. This limitation is the main reason for low recall and precision report by other researchers [9], [22], [23] who later evaluated Ref-Finder.

Silva and Valente [9] developed RefDiff, a tool capable of detecting 13 refactoring types. In contrast to the previous algorithms, RefDiff, for efficiency, only analyses files that are changed, added or deleted between two versions of the program. To find similar entities exist in the two extracted models, RefDiff represents classes and methods as a multiset of tokens where multiplicity of each entity is equal to its number of occurrences in the multiset. As fields do not contain a body, RefDiff considers tokens of statements that directly access each field. To compute the importance of each token, RefDiff employs a variation of the TF-IDF weighting scheme [24] where a low multiplicity of a token is an indicator of a better similarity between two entities. The accuracy and efficiency of RefDiff was found to be better than that of RMiner [25], RefactoringCrawler [6] and Ref-Finder [8]. However, similar to the evaluation performed by Perte *et al.* [8], the evaluation was solely based on *root canal* refactorings.

Recently, Silva *et al.* [11] extended their original tool and introduced RefDiff 2.0,[4] the first multi-language refactoring detection tool. The implemented tool is capable of detecting refactorings in Java, C and JavaScript applications. The main improvement is that the source code is represented as a tree structure [11], and this allows the refactoring detection approach to be independent of any programming language. Our refactoring detection approach is also language-neutral, although it uses a completely different approach to that of Silva *et al.* [11]. We employ a string representation of the source code and rely on relationships between entities (e.g., method invocation, field access, etc.) rather than language-dependent programming instructions (e.g., `while`, `if`, etc.). RefDiff 2.0 is evaluated with 185 open-source Java applications containing 3,248 true refactoring instances, and a high precision (96%) and an acceptable level of recall (80%) is reported. The feasibility of the proposed approach in detecting refactorings in other programming language is also evaluated using a number of small scale experiments, where the precision and recall for C and JavaScript applications ranges from 88% to 91% [11].

Langer *et al.* [26] propose the first tool capable of detecting refactorings on any Ecore-based modelling language. The major benefit of the approach is that it reuses rules defined for specification of refactorings also for detecting them. Therefore, no need to define new queries for detecting new refactorings. The approach, as first step, checks whether the diff patterns of refactoring specifications occurred in any changes applied in the model, and if so a likely candidate refactoring is determined. However, diff patterns do not contain details of refactoring specifications. Therefore, as second step, for each candidate refactoring, pre- and post-conditions of the refactoring are also evaluated. Nevertheless, it is proven that manual and even automated refactorings do not follow strict pre- and post-condition requirements [27]. Therefore, detecting a refactoring only if all pre- and post-conditions are valid makes the approach less effective, and considering weaker conditions results many false positive instances. The accuracy of the proposed approach is investigated using experimental results obtained from one real-world case study containing 141 refactorings, where a high precision (98%) and an acceptable level of recall (70%) is reported.

The majority of existing refactoring detection tools suffer from one important weakness: "*user-provided code similarity thresholds*" [2]. To overcome this weakness, Tsantalis *et al.* [2] develop a refactoring detection tool called RMiner[5] that employs a replacement technique to find similar entities in two extracted models without defining any entity similarity threshold. To improve the efficiency of the algorithm, RMiner only analyses source files which are changed, added or deleted between two versions of the program. It is shown that this technique decreases the number of incorrect code entity matches and consequently improves efficiency of the algorithm [2]. A similar technique is also used in this paper to improve the accuracy and speed of the implemented tool. RMiner is also capable of tolerating unparseable programs which is a weakness of the majority of the aforementioned tools including RefDiff 1.0 [9], Ref-Finder [8], UMLDiff [7] and RefactoringCrawler [6]. The approach presented in this paper is also not dependent on a fully-built version of the system and is capable of tolerating unparseable programs. RMiner is evaluated on 185 open-source Java projects containing 3,188 refactorings, and the authors report a high precision (98%) and recall (87%) for their approach. The superiority of RMiner is also investigated by Tan and Bockisch [23] where RMiner outperforms its competitors: RefactoringCrawler [6], Ref-Finder [8] and RefDiff 1.0 [9]. This superiority has also been observed by Silva *et al.* [11], where RMiner 1.0 (the first version of RMiner) outperforms RefDiff 2.0.

Recently, Tsantalis *et al.* extended their original tool and introduce RMiner 2.0.1[5] [3]. The tool is capable of detecting 40 refactorings types including low-level ones that occur in the method body (e.g., *Inline/Extract/Split/Rename Variable*). The main improvement is in the matching function, where new replacement types and heuristics are

[4]https://github.com/aserg-ufmg/RefDiff

[5] https://github.com/tsantalis/RefactoringMiner

defined. In evaluation the authors compare their tool with existing tools including its predecessor RMiner 1.0 [2], and RefDiff 2.0 [11]. The results, including 7,226 true refactoring instances, show the superiority of the new version of RMiner, where it achieves the best precision (99.6%) and recall (94%) from the tools evaluated. The approach developed in this paper is compared with RMiner 2.0.1. The details of comparison with experimental results along with details of refactoring detection algorithm implemented in RMiner 2.0.1 are presented in the Evaluation Section.

In other recent work, Stevens *et al.* [10] propose a tool-supported approach capable of finding minimum transformations for a desired evolution pattern from changes performed between two versions of a Java program. The proposed approach is based on two graphs: *Change Dependency Graph* (CDG) and *Evolution State Graph* (ESG). Each node in the CDG represents a change, and edges represent dependencies between the changes. As an example, a change that creates a new class must be executed before a change that moves a method to the newly-created class. On the other hand, each node in the ESG contains an abstract syntax tree state and each edge between two nodes show a possible change that can be applied in the source node and results in the target node. Dependencies between changes are also included in the ESG by consulting the CDG during the ESG creation process. At this stage, users using a declarative program querying language, called EKEKO [28], can query the resulting ESG to find desired transformations. However, the use of EKEKO involves a steep learning curve which militates against those with no prior EKEKO experience using it. The proposed approach is not also capable of detecting refactorings involving multiple files because including dependencies across all files results in an unfeasibly large CDG and hence an intractable search space [10].

Krasniqi and Cleland-Huang [12] implement a commit message refactoring detection tool called CMMiner that is capable of detecting 12 refactoring types based on analysing commit logs provided by developers. The proposed approach is evaluated on four Java open source systems and compared with the first version of RMiner [2]. While the results show that relying on commit logs is not enough to detect all applied refactorings, surprisingly CMMiner correctly identifies 10.30% to 19.51% refactorings of different types that were not detected by RMiner [12]. While the majority of refactorings missed by CMMiner resulted from insufficient information in the commit logs about the applied refactorings, the majority of refactorings not detected by RMiner were ones that occurred across multiple commits [12]. As an example, assume a case that an *Extract Superclass* refactoring occurs in one commit and then in the next commit some fields and methods are pulled up to the newly-created class. RMiner cannot detect the applied *Extract Superclass* refactoring as no methods or fields were moved to the newly-created class when RMiner analyses the first commit.[6] On the other hand, the *Extract Superclass* refactoring is not detected in the second commit as the class under consideration is no longer a new class in the this commit. In summary, as concluded by Krasniqi and Cleland-Huang, the achieved results clearly indicate that the use of information in the commit logs along with structural and semantic information contained in the source code can significantly improve current refactoring detection approaches.

As mentioned, the refactoring detection algorithm presented in this paper is based on a differencing algorithm, where refactorings are identified based on differences between the original and refactored programs. The proposed approach, as will be discussed in Section III-C, uses similarity thresholds to match entities (classes, methods, and fields) in the original and refactored programs. While using similarity thresholds negatively affects the accuracy of the approach especially when a large number of *non-refactoring changes* are applied to the refactored entity[7] (see Section IV-C3), it has unique features that allows it to compete with the current state-of-the-art refactoring detection tool: RMiner [3]. The proposed approach employs a simple, but powerful technique based on a string alignment algorithm to detect changes applied in two program versions (see Section III-B), and then uses entities signature and their relationships to match entities different in two programs (see Section III-E). To reduce issues related to similarity thresholds, a novel two-step algorithm that uses refactorings detected in the first round to enhance detection in the second round is introduced (see Section III-C). Furthermore, a feature that distinguishes our tool from other ones is that it is not dependent on any particular programming language, and is able to detect refactorings in any class-based, object-oriented programming language, and potentially even design models such as a UML class diagram. This feature is achieved as the refactoring detection algorithm is not dependent on language-dependent programming instructions (such as `while`, `if`, etc.), and it only relies on entities' signatures and their relationships to match entities in two versions of a program. A detailed description is provided in Section III-C.

III. PROPOSED REFACTORING DETECTION ALGORITHM

Our approach to refactoring detection falls into the 'Differencing Algorithm' approach as described in Section II. Fig. 1 shows an overview of the steps that are followed in our proposed approach. As illustrated, the approach takes as input the original and refactored versions of a program, and

[6] As a necessary condition for *Extract Superclass* refactoring type in RMiner, at least two methods/fields should be pulled up from a subclass to the newly-created class.

[7] As will be discussed in Section IV-C3, the proposed approach critically uses already-identified refactorings in the identification of new refactorings, an issue that not been addressed in many previous approaches, including those proposed by Weißgerber and Diehl [5] and Xing and Stroulia [7].

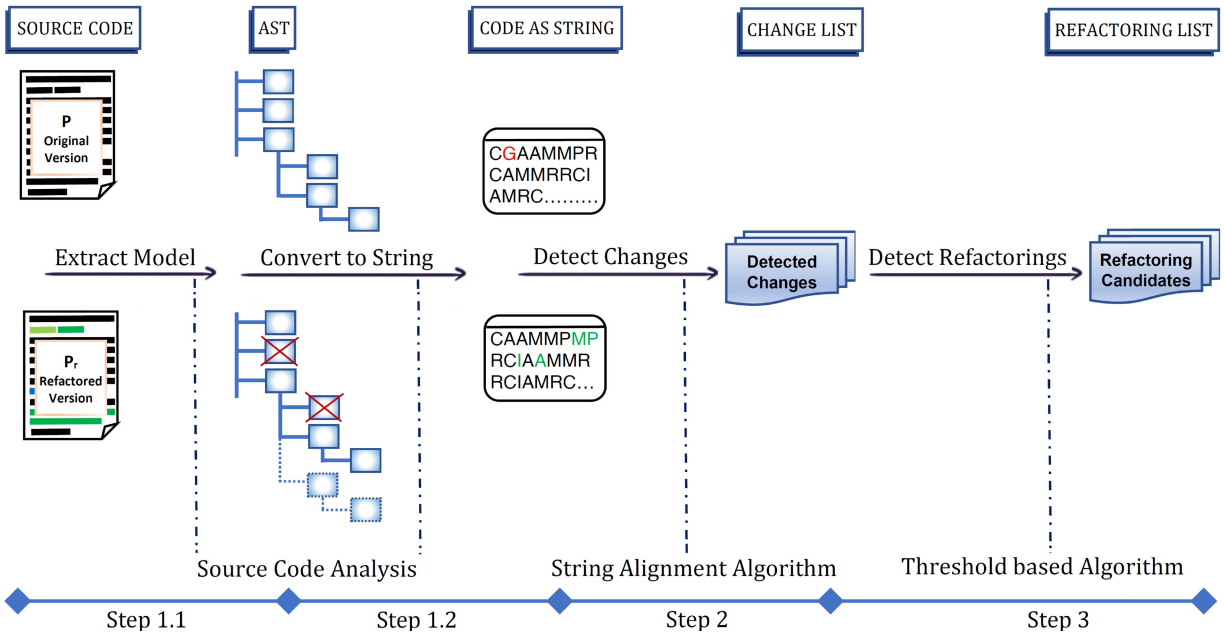


FIGURE 1. Overview of the proposed approach.

generates as output a set of refactoring operations applied between the two input program versions.

The approach is divided into three steps:

1) The code is parsed (Step 1.1) and a model extracted and represented as a sequence of characters (Step 1.2).
2) A change detection algorithm based on a sequence alignment algorithm is used to identify the changes existing between the two input program versions.
3) A threshold based refactoring detection algorithm is used to identify the set of refactoring transformations that represents the evolution from the original version to the refactored one.

Currently, our refactoring detection algorithm is implemented as a tool that operates on Java and C++ applications. We use Spoon[8] and Eclipse CDT[9] as static source code analysis tools to respectively extract information from Java and C++ applications, and represent the extracted information as a sequence of characters. Clearly, Step 1 is language dependent and different code analysis tools must be used for different programming languages. Steps 2 and 3 are not dependant on the programming language of the original source code.

The remainder of this section is structured as follows. A detailed description of how code entities are represented as character sequences, and the employed alignment algorithm to identify changes are presented in Sections III-A and III-B respectively. The refactoring detection algorithm is then discussed in Section III-C. This detection algorithm relies on a number of similarity thresholds, which are described in detail in Section III-D. The algorithm furthermore utilizes a notion of similarity, and this is defined in detail in Section III-E.

[8]https://spoon.gforge.inria.fr/
[9]https://www.eclipse.org/cdt/

A. REPRESENTING THE PROGRAM AS A STRING

The proposed refactoring detection algorithm employs an alignment algorithm to compare program entities in the original and refactored programs with each other. However, this alignment algorithm is based on two input strings. Therefore, initially, program entities such as classes, methods, fields etc. must be represented as a sequence of characters.

To achieve this, we use an approach initially proposed by Kessentini *et al.* [29], and later extended by Hemati Moghadam and Ó Cinnéide [30], [31]. In the employed approach, each entity in the input programs is represented using a specific character as follows: *class (C), interface (I), generalisation relationship (G), attribute (A), method (M), method parameter (P)*, and a call connection between two classes as *(R)*. For example, the representation of class *B* in Fig. 2 is *CGMMPR*. This sequence shows that the class B inherits from another class, contains two methods, where the second method has one parameter, and the class has a coupling relationship with one other class in the program.

In the proposed approach, each character also includes more detailed information depending on the program entity it represents such as name and type for an attribute and the original and called class names for a call connection between two classes. On the other hand, to improve the efficiency of the alignment algorithm, we use a single *R* character to denote a coupling from the original class to another class without counting the number of call connections between the two classes. The reason is that the number of accesses to fields and methods of other classes is usually far greater than the

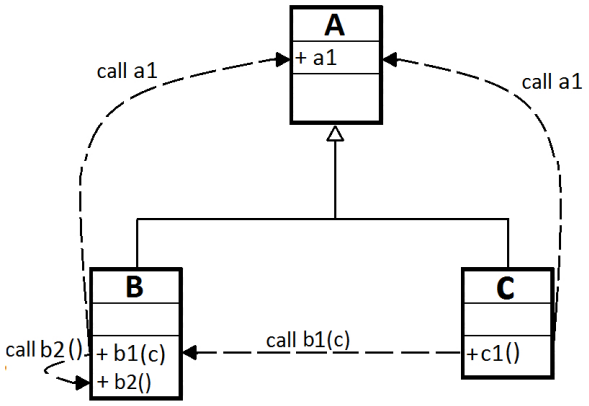


FIGURE 2. The design as a sequence of characters: *CA (class A) CGMMPR (class B) CGMRR (class C).*

number of fields and methods in the class, so representing every method invocation or field reference by one *R* character overemphasises the importance of *R* relationships over the other types when measuring similarity.

The accuracy of employed alignment algorithm depends heavily on order of classes in the string as well as order of method/fields, etc. within a class. To manage this issue, first of all, we have determined a specific order for the features of a class. The order of features of a class in the string is as follows: generalisation relationship, attributes, methods (which themselves are ordered based on the number of their parameters), and finally call connections. For example, as illustrated in Fig. 2, class B is represented as *CGMMPR*. We also sort classes as well as features of classes based on their name. As an example, the exact string representation of class diagram shown in Fig. 2 is CACGMMPRCGMRR, where class A (CA) comes before class B (CGMMPR) and C (CGMRR) in the string, and so on.

B. STRING-BASED SIMILARITY MEASUREMENT

To compare strings representing the input programs, we use a sequence alignment algorithm. A sequence alignment is a way of arranging the sequences in order to determine degree of similarity between them [14], [32]. In this paper, we use *Fast Optimal Global Sequence Alignment Algorithm (FOGSAA)* [14]. The reason that we choose FOGSAA is that while it provides the same result as the best existing alignment algorithms, it is capable of finding the best alignment between two input strings with a lower computational complexity than other global alignment approaches [14].

The algorithm is essentially a branch-and-bound tree algorithm that computes the fitness score for each node in the tree based on its *present* and *future* scores. The present score reflects the current alignment while the future score is an estimation of how well the remaining symbols of the input strings can be aligned. The algorithm returns nodes starting from the root to the leaf of the best branch as the optimal alignment between two input strings [14]. For each pair of characters in the input strings, the alignment algorithm considers three possibilities: *match*, *mismatch*, or *gap* (represented by "−"). If a gap is inserted in one part of a string, the algorithm applies a penalty and it is used to improve the matching of substrings [14].

By way of example, Fig. 3 depicts the sequences $Seq_1 = CAAMM$ and $Seq_2 = CGAM$ being compared using FOGSAA. FOGSAA starts its branch expansion from the root node, and selects the best child among three possible ones according to their fitness score (the summation of present and future scores), and adds other two children in a priority queue according to their fitness score. As illustrated in Fig. 3, the pair [C, C] is selected as the fittest option, and other two pairs namely [C, -], and [-, C] are added in the priority queue according to their fitness score. The process continues with the new node until the end of the first path (i.e. root-to-leaf path), and the resulting path is considered as an initial alignment of the input sequences. In our example, the top path with seven nodes, depicted as solid squares, represents an initial alignment of the input sequences.

In the second round, FOGSAA checks pairs added to the priority queue in order to find whether there is any better alignment, and continues by expanding pairs with a possible fitness score (present score + future score) greater than the best alignment score obtained so far. In our example, pair [C, -] is selected as its future score (4) is greater than the initial alignment score (2), and so a new branch expansion from this node is started (the bottom path). However, during the second round in FOGSAA, a current branch is pruned if at any time its fitness score is not greater than the optimal branch score obtained so far [14]. Therefore, the current branch (the bottom path) stops after two expansions as the nodes that can be extended from pair [-, C] do not have a possible fitness score greater than one achieved so far (2). This process continues until there is no promising pair in the queue, and the best alignment is reported as the optimal alignment. In our example, the best alignment with a fitness score equal to 3 is indicated by the dashed squares.

Clearly, to gain the best alignment it is important to assign the proper values to *match*, *mismatch* and the *gap penalty*. As an example, in Figures 4a and 4b, the value of match and mismatch are set to 1 and -1 respectively. However, different values are assigned to the gap penalty and, as shown, this results in two different alignments between similar input strings. In Fig. 4a, the best alignment when the gap penalty is set to -2, has a similarity score of 0 ($1 - 1 + 1 + 1 - 2 = 0$) while in Fig. 4b where no gap penalty is assigned, the similarity score between input strings is equal to 3 ($1 + 0 + 1 + 0 + 1 + 0 = 3$). For full details of FOGSAA alignment algorithm, interested readers are also referred to the original paper cited above.

C. REFACTORING DETECTION ALGORITHM

In this section, the algorithm we have developed to identify the refactorings applied between two versions of a program is described in detail. First some terms that are used in the algorithm are defined.

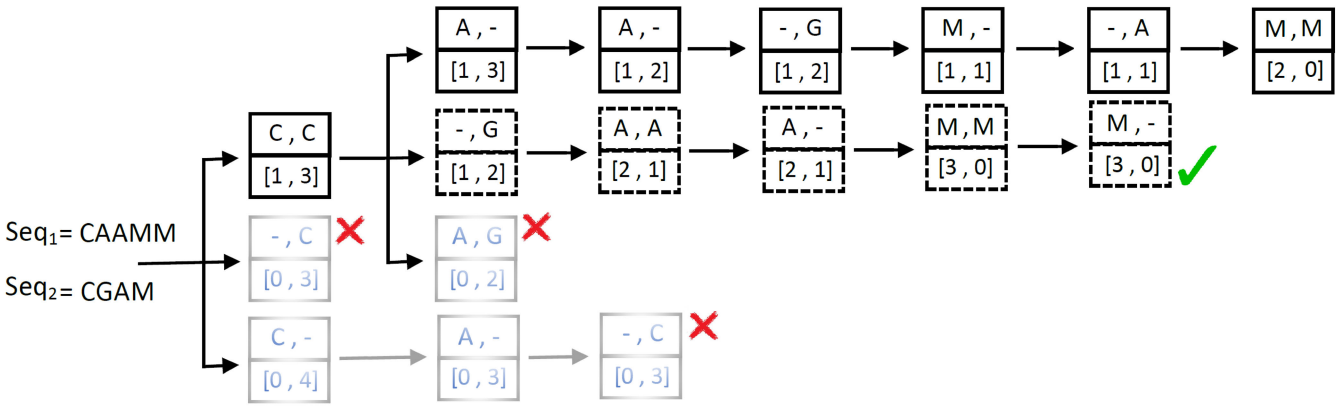
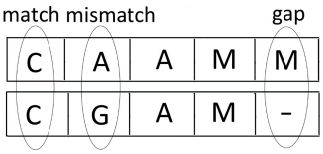
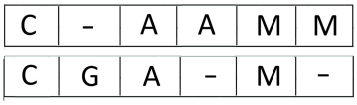


FIGURE 3. **Partially computed FOGSAA tree for the sequences Seq_1 = CAAMM and Seq_2 = CGAM where 1 and -1 values are assigned to the match and mismatch scores respectively and 0 as gap penalty. Each node is annotated on the top with the symbol pairs that are being aligned, and on the bottom with the present score and possible maximum future score [PrS, Fmax]. The top path, shown by solid squares, represents an initial alignment of the input sequences, and the best alignment is illustrated with dashed squares. The 'redX' annotations indicate paths that have been discontinued as their final fitness score will not be greater than the present score obtained so far.**



(a) The best alignment when the gap penalty is set to -2.



(b) The best alignment when the gap penalty is set to 0.

FIGURE 4. **The best alignments for Seq_1 = CAAMM and Seq_2 = CGAM with different gap penalty.**

- **Entity**: Represents a program element including *class* (including nested classes), *interface*, *enum*, *method* (including constructors), and *field* in programming languages.
- **Matched Entities**: A list containing pairs of matched entities where an entity in the original program is matched with an entity with similar type in the refactored program. A matched entity can be a *true* or a *false* instance. A false instance happens when the algorithm incorrectly matched two entities.
- **Unmatched Entity**: An entity in the original program or in the refactored program which has no match in the other program is called an *unmatched entity*. An unmatched entity can be a *true* or *false* instance. A true instance happens when the entity is actually deleted/added in the program, and a false instance happens when the entity is not deleted/added, but the algorithm cannot find a match for the entity in the corresponding program. Matched and unmatched entities are initially identified based on the best alignment sequence.
- **removedEntities**, and **addedEntities**: Two sets that contain **initial unmatched entities** in the original and refactored programs respectively. They are identified based on differences in the best alignment sequences.
- $\mathbf{U_{original}}$, and $\mathbf{U_{refactored}}$: Two sets which respectively contain **final unmatched entities** in the original and refactored programs. These two sets are actually subsets of *removedEntities* and *addedEntities* respectively.
- **Support**, and **Confidence thresholds**: The matching algorithm presented in this paper uses similarity thresholds to match entities in the original program with ones in the refactored program. The entity matching algorithm runs in two rounds, and in each round different similarity threshold values are used. In fact, entity similarity thresholds used in the first round of the matching algorithm (support thresholds) are weaker than confidence thresholds used in the second round. Further details presented in Section III-D.

1) REFACTORING DETECTION ALGORITHM IN DETAILS

The proposed refactoring detection algorithm is illustrated as pseudocode in Algorithm 1. As illustrated, the algorithm takes as input the original and refactored versions of the input program which are both represented as a sequence of characters. The algorithm also receives similarity thresholds (*support*, and *confidence* similarity thresholds) as last input parameter, and produces as output the set of refactorings applied between the two input program versions. As illustrated, the proposed algorithm is divided in three main steps: (i) *inconsistency detection* (Line 1), (ii) *entity matching* (Line 6), and (iii) *refactoring detection* (Line 7). While the first step is done once, the last two steps (ii, and iii) are

executed in two rounds (Lines 6 to 9). A detailed description of each step is provided as follows.

Step 1. Inconsistency Detection: The goal of this step is to recover *initial unmatched entities* between the two versions of the input program. This step is based on the FOGSAA algorithm. As first step, FOGSAA is applied to the string representation of the original and refactored versions of the input program, and then a gap or mismatch character in the best resulting alignment sequences is classified as an unmatched situation. The implemented alignment algorithm compares the *type* (e.g., class, field, etc.) as well as the *name* of entities included in the input sequences to decide about their similarities. As an example, Fig. 5 shows the best alignment sequences that are derived from the two versions of the program illustrated in Fig. 6. As shown, a gap in the second position of Seq_1 indicates an unmatched field in Seq_2 (representing the field *counter* in *class_A* in the refactored program). Where a class in one program has no similar one in other program, all entities in that class are considered as unmatched entities in that program (e.g. entities in the second (CA) and third parts (CGMPR) in Seq_1 and Seq_2 respectively).

$$\begin{aligned} Seq_1 &= \text{C} - \text{A M M P} \mid \text{C A} \mid - - - - - \\ Seq_2 &= \text{C A A M M P} \mid - - \mid \text{C G M P R} \end{aligned}$$

FIGURE 5. The best alignment for the original and refactored version of the program shown in Fig. 6.

Note that the implemented algorithm also determines the fourth position in two sequences to be an unmatched case. While the types of entities in this position are equal (i.e., method), their names are different (*foo()*, and *getCounter()* in Seq_1 and Seq_2 respectively.) Comparing both type and name of entities also allows the algorithm to correctly detect the difference between *class_B1*, and *class_B2* (the seventh and ninth positions in Seq_1 and Seq_2 respectively), and inserts a gap character in these positions in two sequences.

The result of the first step as illustrated in Algorithm 1, is two sets: *removedEntities*, and *addedEntities* which contain entities in the original and refactored programs respectively for which the alignment algorithm could not find a match. Note that the implemented alignment algorithm compares each class in the original program with *its similar one* in the refactored program to determine their differences.[10] Therefore, it is only capable of detecting classes different in two programs and also differences in classes that exist in both programs. It is not capable of matching entities different in both programs. For example, in Fig. 5, the FOGSAA algorithm cannot determine that the field *counter* in *class_B1* (eighth position in Seq_1) should be matched with the field *counter* in *class_A* (second position in Seq_2) or *class_B1*, and *class_B2* are the same.

Step 2. Entity Matching: To cover this issue, in the second step of the algorithm, the similarity of each entity in *removedEntities* is examined with entities of the same type in *addedEntities*, and entities with a similarity value higher than the input similarity threshold considered to be the same. In a case that an entity is matched with more than one entity in the corresponding program, the best match with the highest similarity value is selected. The only exceptions are when an entity is pushed down to multiple subclasses or when a pull up refactoring is called on more than one similar entity in different subclasses. In these cases, the matching algorithm allows one-to-many and many-to-one matching respectively. On the other hand, in a case that an entity has no similar one in the other program, it is considered as unmatched entity in that program. Note that non-matching entities can occur due to refactoring or non-refactoring changes. For example, a newly-created field is an unmatched entity due to non-refactoring changes, while an unmatched method in the refactored program which is extracted using an *Extract Method* refactoring is the result of a refactoring.

The matching of distinct entities in the original and refactored programs is the most important function used throughout the refactoring detection algorithm used in this paper. In fact, incorrect matching of entities at this stage greatly reduces the accuracy of the refactoring detection algorithm. Therefore, to reduce the possibility of erroneous matches, a **two-step** conservative approach is employed (Lines 5 to 9 in Algorithm 1). In the entity matching algorithm, the values for the similarity thresholds are increased by about 30% in the second round (see Table 3). In fact, the second round has a *more strict* match conditions than the first round (compare lines 4 and 8 in Algorithm 1).

The reason for using this technique is as follows: The goal of the *first round* is to identify entities that have been *removed* or *added* in the program. These entities with a high probability have little similarity with unmatched entities in their corresponding program, and even low values for the similarity threshold will not register them as being similar. Therefore, in the first round, with a high probability we can determine entities which are actually deleted/added in the program.[11] Another goal in the first round is to identify the entities that have been *renamed*. The initial assumption is that a renamed entity in the original program has a high resemblance to an entity with a different name in the refactored program. Therefore, even low values for the similarity threshold will help to find renamed entities. If the program includes small changes, the other refactoring types detected in the first round are also valid (for example when a method is moved to another class using a *Move Method* refactoring). However, when the

[10] As mentioned in Section III-A, classes and their features in both input strings are sorted based on their names. This increases the chance that similar classes in two programs are compared with each other during alignment.

[11] Our evaluation shows that extracted and inlined methods also have minor similarity with other unmatched methods. Consequently, the algorithm detects these methods as deleted or added entities. Later, in Step 3 of Algorithm 1, we will discuss how determining a specific order in detecting refactorings helps to detect these two refactoring types correctly.

Algorithm 1 : Refactoring Detection Algorithm

Input: The original and refactored versions of the input program represented as two sequences of characters: Prg_1 and Prg_2.
Input: Support and confidence similarity thresholds (defined in Table 3).
Output: Set of refactoring operations applied between two versions of the input program.

Function RefactoringDetection(Prg_1, Prg_2, similarityThresholds): Detected Refactorings

Step 1: Extract initial unmatched entities by applying FOGSAA on the two versions of program (Prg_1, and Prg_2)
1: inconsistencyDetection(Prg_1, Prg_2): removedEntities, addedEntities
2: counter = 1
3: detectedRefs = ∅
4: thresholds = similarityThresholds.supportThreshold
5: **while** (counter ≤ 2) **do**

Step 2: For each entity in removedEntities, where possible, find its corresponding entity in addedEntities.[**]**
6: entityMatching (removedEntities, addedEntities, thresholds, detectedRefs): matchedEntities, $U_{original}$, $U_{refactored}$

Step 3: Extract refactorings using sets resulting from Step 2, and according to some predefined rules.[‡‡]
7: detectRefactorings(matchedEntities, $U_{original}$, $U_{refactored}$): detectedRefs
8: thresholds = similarityThresholds.confidenceThreshold /* About 30% higher than the supportThreshold. */
9: counter = counter + 1
end while
10: **return** detectedRefs
End Function

[**] See Algorithm 2 for more detail, [‡‡] See Tables 1 and 2 for more detail.

```java
public class Class_A {

    private int iterator;

    public void foo() {
        //TODO
    }

    public Class_A(int iterator) {
        this.iterator = iterator;
    }
}

public class Class_B1 {
    public int counter = 1;
}
```

(a) The original program (as string: CAMMP CA)

```java
public class Class_A {
    private int counter;
    private int iterator;
    public int getCounter() {
        return counter;
    }
    public Class_A(int iterator) {
        this.iterator = iterator;
        this.counter = 1;
    }
}

public class Class_B2 extends Class_A {

    public Class_B2(int iterator) {
        super(iterator);
    }
}
```

(b) The refactored program (as string: CAAMMP CGMPR)

FIGURE 6. Some of the changes between the two versions: *Class_B1* is renamed to *Class_B2* and inherits from *Class_A*. Field *counter* in *Class_B1* is pulled up to *Class_A*. A new constructor is created in *Class_B2*, and the methods *foo()*, and *getCounter()* are respectively deleted from, and added to, *Class_A*.

program involves a lot of changes (including refactoring and non-refactoring changes), there is a possibility of incorrect matches.

To reduce the possibility of erroneous matches, **in the second round, the similarity threshold values are increased** (Line 8 in Algorithm 1) and the matching function is called

again for all entities included in the *removedEntities* and *addedEntities* sets *even for those entities that had a high similarity in the first round*. However, when comparing two entities, **refactorings detected in the first round are used** to improve the accuracy of the matching process. As shown in Algorithm 1, the implemented entity matching function (*entityMatching*) takes as last parameter the list of detected refactorings (*detectedRefs*). This list is empty in the first round (Line 3). However, in the second round, it contains refactorings detected in the previous round. Overall, the refactorings identified in the first round provide extra information that enables higher threshold values to be used. It is worth noting that while the refactoring list proposed in the first round is likely to contain false positive instances, our experiments show that the benefit of the approach is much greater than the negative effect due to false positive refactorings detected in the first round.

Initial detected refactorings are actually used when two entities are compared and their similarity based on *their shared relationships* is measured. To clarify the matter, the process is explained with some examples. Assume that two methods are compared with each other by the matching algorithm. The algorithm, apart from comparing the name of the methods, also *examines the similarity of their relationships* (methods and fields that are called by these two methods and also methods that call these two methods). However, when the related entities are themselves refactored, the accuracy of relationship analysis depends heavily on refactorings detected in the first round. In fact, relationships that are due to entities that are identified in the first round as deleted from the original program or as new in the refactored program are ignored and not counted as differences between the two methods. In addition, related entities that have been renamed are considered identical.

The algorithm also supports changes made by other refactorings. For example, when a field or method called by methods under investigation is moved to another class in the refactored design, this class change is considered when measuring relationships similarity. As another example, when a method is extracted from methods under investigation, the extracted method is not considered as a difference between the methods.

For clarity, the process is explained with an example that was observed during the evaluation performed in this paper. The observed case is simplified, and presented as a design shown in Fig. 7. As illustrated, three refactorings and three non-refactoring changes occur in the input program as follows: Class *A* is Renamed to *B*, Field *name* is Renamed to *fullName*, and method *getName()* is Renamed to *getFullName()*. Furthermore, method *xyz()* is Removed from the class *A* and method *getAge()*, and field *age* are Added in the class *B* (non-refactoring changes). In this example, the FOGSAA algorithm correctly detects that nine changes exist between the original and refactored designs.

In the following, we examine a scenario that may occur during the matching of the unmatched entities in two designs.

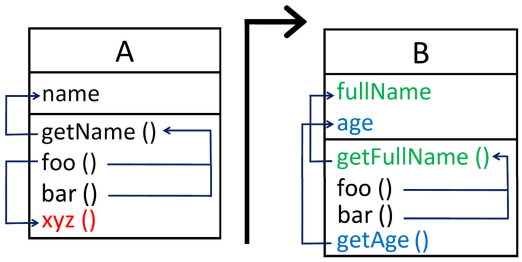


FIGURE 7. The original and refactored designs of a simplified case observed during evaluations. Arrows in classes show field access and method invocation relationships.

As first step, assuming field *name* in the original design is compared with fields *fullName* and *age* in the refactored design. Since there is no similarity in their names as well as their relationships, the algorithm regards the field *name* as being completely different from the other two fields in the refactored design. The same conditions are considered when comparing different methods in two designs. However, in this case, methods *getName()* and *getFullName()* are detected as being the same. In fact, these two methods have different names, but their relationships (with methods *foo()*, and *bar()*) completely match. Therefore, the refactoring ''Rename Method *getName()* to *getFullName()*'' is detected correctly. As a final step, classes *A* and *B* are compared. The algorithm detects these classes to be different ones. This happens as the names of the classes are different, and there is not strong similarity in methods and fields defined in these classes (only methods *foo()* and *bar()* are similar in these classes).

As observed, in the first round, one refactoring was detected correctly, and while no refactoring was detected incorrectly, two valid refactorings were not detected by the algorithm. However, it is clear that depending on the changes have been made in the program, more false positives or false negatives refactorings may be seen in the first round. Therefore, *in the second round, to reduce false positive instances, the similarity threshold values are increased, and to reduce false negatives cases, the refactorings identified in the first round are used*.

In our example, the *Rename Method* refactoring detected in the first round increases the similarity of two classes and the algorithm correctly detects the refactoring ''Rename Class A to B''. The rename method refactoring also increases the relationship similarity between fields *name* and *fullName* and the algorithm correctly identifies ''Rename Field *name* to *fullName*''.[12]

As far as we know, detecting refactorings in two rounds as presented in this paper is not considered by any previous research works. However, as will be discussed in Section III-E, another novelty of our approach is that the matching algorithm relies on entities name and their relationships, and does not rely on any language-dependent programming instructions (e.g., `while`, `if`, etc.). This allows

[12]The importance of the similarity of entities' names and their relationships is discussed in section III-D.

TABLE 1. Refactorings detected by the proposed approach.

No.	Class-Level Refs.	Description
1	Rename Class	Change the name of a class to a new name, and update its references.
2	Move Class	Move a class to a more relevant package.
3	Extract Class	Create a new class and move relevant features to the new class.
4	Inline Class	Move all features of a class to another class and delete the empty class.
5	Extract Subclass	Add a new subclass to a class and push down relevant features to it.
6	Extract Superclass	Add a new superclass to class and pull up relevant features to it.
7	Collapse Hierarchy	Move all features of a class to its super or sub class and delete it.
8	Extract Interface	Create a new interface from selected classes, and make the selected classes inherit from the newly created interface.
	Field-Level Refs.	
9	Rename Field	Change the name of a field to a new name, and update its references.
10	Push Down Field	Move a field from a class to those subclasses that require it.
11	Pull Up Field	Move a field from some class(es) to their immediate superclass.
12	Move Field	Move a field from a class to another one which uses the field most.
	Method-Level Refs.	
13	Rename Method	Change method name to a new name, and update its references.
14	Push Down Method	Move a method from a class to those subclasses that require it.
15	Pull Up Method	Move a method from some class(es) to their immediate superclass.
16	Move Method	Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it.
17	Extract Method	Move a code fragment in an existing method to a newly created method and replace the old code with a call to the new method.
18	Inline Method	Replace calls to the method with the method's content and delete the method itself.
19	Change Method Parameters	Including insert, remove, and reorder of method's parameters, and also change method's parameters type.
	Composite Refs.	
20. Extract & Move Method, 21. Move & Inline Method, 22. Move & Change Method Parameter, 23. Move & Rename Class, 24. Move & Rename Method, 25. Move & Rename Field, 26. Inline to an Inline Method, 27. Extract from an Extract Method.		

us to develop a multi-language refactoring detection tool. Later in section III-E, the matching algorithm is discussed in more detail. We will also provide more detailed information about similarity thresholds used in the matching algorithm in Section III-D. However, first, the final step in the refactoring detection algorithm is discussed.

Step 3. Refactoring Detection: Deciding about applied refactorings is the third step in Algorithm 1. Currently, the implemented tool supports the detection of 27 refactoring types (including composite ones) as illustrated in Table 1. These form a representative subset of the common set of refactorings proposed by Fowler [33].

The implemented function (Line 7 in Algorithm 1) takes as input three sets, all produced from Step 2. While the first input set contains pairs of entities mapped from the original program to the refactored program (*matchedEntities*), the second and third input sets contain non-matching entities in the original and refactored programs respectively ($U_{original}$ and $U_{refactored}$). The refactoring detection function, using this information and based on a set of predefined rules as given in Table 2, determines what refactorings are deemed to have been applied. For example, as illustrated in Table 2, assuming $\underline{o}$ and $\underline{r}$ are the original and refactored revisions of a program respectively. Then, an *Extract Class* refactoring is detected if *(i)* there is a class such as $\underline{c_r}$ in the refactored program that does not match any class in the original program ($c_r \in U_r$), *(ii)* there is a class such as $\underline{c_k}$ in the refactored program that has no inheritance relationship with class $\underline{c_r}$ ($\neg$inheritanceHierarchy(c_k, c_r), and (iii) at least two fields or methods are moved from class $\underline{c_k}$ to the newly created class (movedEntities(c_k, c_r) ≥ 2).[13]

As another example illustrated in Table 2, a *PushDown* refactoring is detected if *(i)* there is an entity such as $\underline{e_o}$ in the original program that is matched to at least one entity such as $\underline{e_r}$ in the refactored program and *(ii)* the contain class of $\underline{e_o}$ is a superclass of its matched entities.

Note that, we do not include the rules defined to detect composite refactorings in Table 2. However, these refactorings are detected by combining rules of included refactorings. As an example, an *Extract and Move Method* refactoring is detected if conditions for both *Extract* and *Move Method* refactoring types are valid for a newly created method.

D. SIMILARITY THRESHOLDS

As discussed in section III-C1, refactoring detection algorithm (illustrated as Algorithm 1) is run in two rounds, and different thresholds are used in each round of the algorithm. We name thresholds used in the first round as *Support Thresholds* and those used in the second round as *Confidence Thresholds*. As shown in Table 3, each type itself contains three parts: *entity similarity threshold*, *name importance threshold*, and *relationship importance threshold*. In the following, these topics are discussed with more details.

[13] In Evaluation section, we will discuss the reason for the condition of moving at least two methods or fields to the extracted class, and we will explain how this condition can affect the accuracy of this refactoring type.

TABLE 2. Refactoring detection rules.

Refactoring type	Rule
Rename Entity e_o to e_r	$matching(e_o, e_r) \wedge e_o.name \neq e_r.name$
Move Entity e_o to e_r	$matching(e_o, e_r) \wedge e_o.class^{\sim}$!= $e_r.class \wedge \neg inheritanceHierarchy(e_o.class^{\sim}, e_r.class)$
Pull Up Entities $e_{o.1}$ ... $e_{o.n}$ to e_r	$\forall k \in \{1...n\}\ (matching(e_{o.k}, e_r) \wedge superType(e_r.class, e_{o.k}.class^{\sim}))$
Push Down Entities e_o to $e_{r.1}$ ... $e_{r.n}$	$\forall k \in \{1...n\}\ (matching(e_o, e_{r.k}) \wedge superType(e_o.class^{\sim}, e_{r.k}.class))$
Extract Method m_r from m_o	$m_r \in U_r \wedge calls(m_o^{\sim}, m_r) \wedge sim(m_o - m_o^{\sim}, m_r) \geq thrshold$
Inline Method m_o in m_r	$m_o \in U_o \wedge calls(m_r^{\sim}, m_o) \wedge sim(m_r - m_r^{\sim}, m_o) \geq thrshold$
Change Parameters of $m_o.p_1$ to $m_r.p_2$	$matching(m_o, m_r) \wedge p_1 \neq p_2$
Inline Class c_o	$c_o \in U_o \wedge ((\exists\ c_k \in o \wedge \exists\ c_k^{\sim} \in r) \mid (\neg inheritanceHierarchy(c_o, c_k) \wedge movedEntities(c_o, c_k^{\sim}) \geq 2)$
Collapse Hierarchy c_o	$c_o \in U_o \wedge ((\exists\ c_k \in o \wedge \exists\ c_k^{\sim} \in r) \mid (inheritanceHierarchy(c_o, c_k) \wedge movedEntities(c_o, c_k^{\sim}) \geq 1)$
Extract Class c_r	$c_r \in U_r \wedge (\exists\ c_k \in o \mid (\neg inheritanceHierarchy(c_k^{\sim}, c_r) \wedge movedEntities(c_k^{\sim}, c_r) \geq 2)$
Extract Superclass c_r	$c_r \in U_r \wedge ((\exists\ c_k \in o \wedge \exists\ c_k^{\sim} \in r) \mid superType(c_r, c_k^{\sim}) \wedge movedEntities(c_k^{\sim}, c_r) \geq 1)$
Extract Subclass c_r	$c_r \in U_r \wedge ((\exists\ c_k \in o \wedge \exists\ c_k^{\sim} \in r) \mid superType(c_k^{\sim}, c_r) \wedge movedEntities(c_k^{\sim}, c_r) \geq 1)$
The data structures and functions used are defined as follows: - **o** and **r**: Assuming o and r are the original and refactored versions respectively. - **U_o** and **U_r**: List of unmatched entities in the program o and r respectively. - **matching(e_o, e_r)**: returns true if pair (e_o, e_r) exists in the matchedList. - **superType(c_1, c_2)**: returns true if class c_1 is a direct or indirect superclass of c_2. - **inheritanceHierarchy(c_1, c_2)**: superType(c_1, c_2) $\vee$ superType(c_2, c_1). - **movedEntities(c_1, c_2)**: returns the number of fields and methods moved from class c_1 to class c_2. - **calls(e_1, e_2)**: returns true if entity e_1 calls directly or indirectly entity e_2. - **sim(e_o, e_r)**: returns the similarity value between entities e_o and e_r. - **$e_o^{\sim}$**: returns an entity in the program r that matches with entity e_o. **$e_r^{\sim}$**: returns an entity in the program o that matches with entity e_r.	

1) ENTITY SIMILARITY THRESHOLD

Entity similarity threshold actually shows the least similarity that two entities (e.g., classes, fields, etc.) must have in order to identify them as the same. As shown in Table 3, the *entity similarity threshold* used as support one is about 30% less than one used as confidence one (0.5 vs. 0.7). As discussed in section III-C1, the reason for this increase is that to prevent incorrect entity matching and so reduce false positive refactorings detected in the first round of refactoring detection algorithm. It is worth mentioning that we used the same entity similarity threshold for all entities (including class, method, etc.). In other word, different thresholds are not used for different entity types.

2) NAME IMPORTANCE THRESHOLD vs. RELATIONSHIP IMPORTANCE THRESHOLD

The similarity between two entities are measured based on similarity of their *names* as well as their *relationships*. In the proposed approach, the importance of entities name and their relationships varies. As shown in Table 3, a different value is assigned to the name and relationship importance thresholds. For example, in support thresholds, the value assigned to name importance threshold is 0.4 while it is 0.6 for relationship importance threshold (relationship importance threshold = 1 - name importance threshold). On the other hand, these values are also different in the support and confidence thresholds. As show in Table 3, for example, the name importance threshold as support one is 0.4 while it is 0.3 as confidence one.

TABLE 3. Experimentally-evaluated similarity thresholds.

	Entity Similarity Threshold	Name Importance Threshold*
Support thresholds	0.5	0.4
Confidence thresholds	0.7	0.3
* Relationship Importance Threshold = 1 - Name Importance Threshold		

It was discussed that the goal in the first round of the algorithm is to identify entities that have been renamed, removed or added in the program. In the first round, the importance of the similarity of names and relationships are specified 0.4 and 0.6 respectively. On the other hand, the value for the class/field/method similarity threshold is considered 0.5. This means that, for example, two methods with different names (-0.4), but the similarity of relationships above 84% are likely to be the same ($(0.6 * 0.84) > 0.5$). This allows to identify Rename Class/Method/Field refactoring candidates

whose relationships have been slightly changed. On the other hand, class/method/field that have been removed or added to the program, in a case that there is no similar names in the corresponding design, are also easily detectable.[14]

In the second round of algorithm, the importance of similarity of names decreases (0.3) while the importance of the relationships between the entities increases (0.7). However, as discussed in section III-C1, since the refactorings detected in the first round are used when comparing the relationships of entities under investigation, a reduction in false negatives refactorings is expected. Note that meanwhile increasing the *entity similarity threshold* helps to reduce false positives instances. A detailed description of how similarity of different entities (classes, fields, etc.) based on name and relationship similarities is measured is provided in the section III-E.

3) CALIBRATION OF SIMILARITY THRESHOLDS

Obviously, the selected threshold values affect the precision and recall of the algorithm. Therefore, to improve the accuracy of the algorithm, we employed the approach used by RefDiff [9] to calibrate similarity thresholds. We have used 10 commits of 514 project commits presented by Tsantalis *et al.* [3] to calibrate similarity thresholds. We selected the projects based on two criteria: (i) the project contains the most refactoring and non-refactoring changes and (ii) includes almost all refactoring types supported by our tool. It is worth mentioning that all refactorings detected in these projects are confirmed by the project developers, or have been manually reviewed and approved by Silva *et al.* [25], or Tsantalis *et al.* [3]. We also used a dataset of seed refactorings contains 200 refactorings which all designed by the authors of this paper to evaluate edge cases in C++ and Java applications.[15]

Note that we calibrate thresholds only based on a small number of commits (10) and a number of seed refactorings. These thresholds remain unchanged throughout our experiments involving 514 commits of 185 Java applications, and four C++ applications, which gives confidence that they do not need to be adjusted on a per-application basis.

E. SIMILARITY MEASUREMENT ALGORITHM IN DETAIL

The refactoring detection algorithm is discussed in Section III-C. As mentioned, the approach employs a similarity measurement algorithm to determine the similarity of two entities (Step 2 in Algorithm 1). However, we did not provide much detail about the similarity measurement algorithm other than that it compares entities based on their names as well as their relationships, and determines two entities to be the same if they have a similarity value more than the entity similarity threshold. In this section, the similarity measurement algorithm is explained in detail.

[14]It is assumed that it is unlikely that a deleted entity in the original program have a high relationship similarity with an entity with a different name in the refactored program. The same assumed for a newly added entity.

[15]These programs are different than those used to evaluate the feasibility of the refactoring detection approach with C++ applications (section IV-D).

Before delving into detail, it is useful to recall two points. Firstly, the algorithm is based on information included in the string representation of the input program. Secondly, the algorithm compares entities based on their names and their relationships with other entities, and it is not dependent on any language-specific programming instructions (such as `while`, `if`, `switch`, etc.). These two factors make it possible to develop a refactoring detection tool that is not dependent on any programming language.

The similarity function is presented in Algorithm 2. As illustrated, the algorithm receives as input two entities that should be compared, the similarity threshold values, the list of current matching entities, and the list of refactorings identified in the previous round. The algorithm produces as output the similarity value of the input entities normalized as a value between 0 and 1. To increase the speed of the algorithm, as the first step, if the input entities were already compared with each other in the current round, their measured similarity value is returned immediately (Lines 1 and 2). Otherwise, depending on the type of input entities (class, field or method), the appropriate part of the algorithm, as discussed below, is executed.

1) CLASS SIMILARITY

The similarity between two classes is measured based on similarity between methods and fields defined in the classes (Lines 3 to 9). In fact, methods and fields which have the best similarity value and their similarity is greater than the determined similarity threshold are identified as corresponding entities (Line 8). After comparing entities defined in two classes, the ratio of the number of matching entities to the sum of the entities defined in the two classes is returned as the similarity of the input classes (Line 9).

It is worth mentioning that when comparing classes, the fields and methods that have been removed from the class in the original program or added to the class in the refactored program, both using non-refactoring changes, are considered as differences between two classes. Otherwise, the algorithm would find a deleted class, and a completely new class to be strongly similar. However, changes resulting from refactorings detected in the first round are taken into account. For example, changes in parameters of a method or its name respectively due to *Change Method Parameter* and *Rename Method* refactorings are considered when comparing two methods. As another example, where a method in the original class is moved to another class using a *Move Method* refactoring, this is not considered as a difference between two corresponding classes. To achieve this, as a first step and before comparing fields and methods of the input classes, a pre-processing is performed on fields and methods of classes and their main entities are retrieved (Line 4).

2) FIELD/METHOD SIMILARITY

In a case that the input entities are fields, their similarity is measured based on the similarity of their names as well as the similarity of their relationships with other entities. As a first

Algorithm 2: Measure Similarity between two input entities

Input: Two program entities with similar type (class, field, etc.) represented as **entity1** and **entity2**
Input: Similarity thresholds: **entitySimilarityThreshold**, and **nameImportanceThreshold**
relationshipImportanceThreshold = 1 - nameImportanceThreshold
Input: List of matched entities in the current round represented as **matchedList**
Input: List of detected refactorings in the previous round represented as **detectRefs**
Output: Similarity value of **entity1** and **entity2** normalized as a value between 0 and 1.

Function MeasureSimilarity (entity1, entity2, thresholds, matchedList, detectRefs) : Double

```
    //If entity1 and entity2 are already matched, returns their similarity value.
1   if (matchedList.contains(entity1, entity2)) then
2       return matchingValue(entity1, entity2)

    Case1: Measure similarity between two classes: entity1 and entity2 represented as c1 and c2.
3   if (entity1 represented as c1 is a class) then
4       entityList1 , entityList2 = preprocessClass(c1, c2, detectRefs)
5       foreach (e1 ∈ entityList1) do
6           foreach (e2 ∈ entityList2 where e1.type == e2.type) do
7               temporaryEntityMatchedList.add(e1, e2, MeasureSimilarity(e1, e2, thresholds, matchedList, detectRefs))
8       matchingSet = findBestMatch(temporaryEntityMatchedList, entitySimilarityThreshold)
9       return ((matchingSet.size() * 2) / (entityList1.size() + entityList2.size()))

    Case2: Measure similarity of two fields or methods: entity1 and entity2 represented as e1 and e2.
10  if (entity1 represented as e1 is a field or method) then
11      if (not(haveSimilarName(e1, e2))) then
12          nameImportanceThreshold = 0
13      relationshipSimilarity = intersection(callers(e1), callers(e2)) / union(callers(e1), callers(e2))

    Case2.1: Measure similarity between two methods: entity1 and entity2 represented as m1 and m2.
14  if (entity1 represented as m1 is a method) then
15      relationshipSimilarity += intersection(invokers(m1), invokers(m2)) / union(invokers(m1), invokers(m2))
16  return nameImportanceThreshold + (relationshipImportanceThreshold * normalize(relationshipSimilarity))
```

End Function

The functions used are defined as follows:
matchingValue(e_1, e_2): returns similarity value between e_1 and e_2.
preprocessClass(c_1, detectsRef): returns fields and methods in class c_1.‡‡
haveSimilarName(e_1, e_2): returns true if (e_1.name = e_2.name $\vee$ isRenamed(e_1, e_2)) $\wedge$ compatibleSignature(e_1, e_2)
isRenamed(e_1, e_2): returns true if e_1 is renamed to e_2 based on refactorings detected so far.
compatibleSignature(m_1, m_2): return true if methods m_1, and m_2 have similar input parameters.‡‡
callers(e_1): returns **list** of methods that call e_1.**
invokers(m_1): returns **list** of methods and fields that m_1 calls.**
union(set1, set2) returns number of the entities that are at least in one of the two sets.‡‡
intersection(set1, set2) returns number of the similar entities that are in both sets.‡‡
‡‡ All refactorings detected in the first round are used to improve the accuracy of the function.
** Exclude methods and fields which are deleted from the original program or added in the refactored program.

step, the name of entities are compared with each other (Lines 11 and 12), and if the entities do not have an identical name or the first entity is not renamed to the second one, the name similarity threshold for those entities is set to zero.[16] When comparing two fields, the similarity of methods that call the fields is also measured (Line 13).

However, when examining two methods, in addition to the points mentioned, the similarity of fields and methods called by these two methods are also examined (Lines 14 and 15). The previously identified refactoring and non-refactoring changes are also used here to improve the accuracy of the algorithm (see *compatibleSignatures*, *callers*, *invokers*, *union*, and *intersection* functions in Algorithm 2). As a final step, the relationship similarity of the input entities is normalized, and the similarity between two entities is computed based on similarity of their names and relationships (Line 16).

An important issue is that the formula used to measure call similarity between two fields or methods (Line 13) might not work properly when the entity has *more than one* matching candidate in its hierarchy structure in the corresponding program. This case happens when an entity is pushed down to multiple subclasses or when a pull up refactoring is called on more than one similar entity in different subclasses.[17] The problem is described with an example.

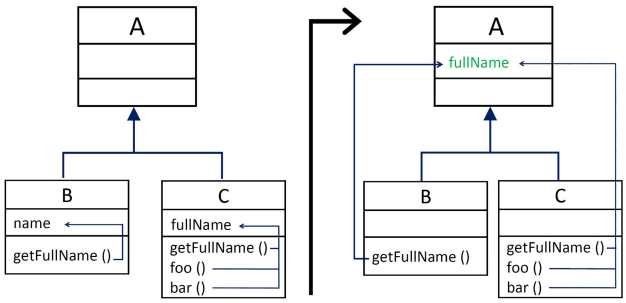


FIGURE 8. The original and refactored designs of a program. Arrows inside classes show field access relationships.

In Fig. 8, field *name* in class B is renamed to *fullName* and then both fields in classes B and C are pulled up to their direct superclass. In this example, there is only a small similarity between field *name* in the original class B and field *fullName* in the refactored class A. In fact, as well as having different names, of the four methods that call field *fullName* in class A, method *getFullName* in class B is the only common entity between these two fields. Thus, these fields are not identified by the matching algorithm as the same and consequently the applied refactorings *Rename Field name to fullName*, and *Pull Up field name to class A* are not detected by the algorithm. The problem arises from the fact that the relationships resulting from the methods in class C are considered when comparing fields *name* in class B with the field *fullName* in class A. To overcome this issue, we need to change the denominator of the formula presented in Line 13 in Algorithm 2.

In fact, in the case of a pull up refactoring, only related methods in the original design should be compared as expressed in Eq. 1, and in a push down case only related methods in the refactored design should be compared (the denominator of Eq. 1 should be changed to $callers(e_2)$). Using this technique, fields *name* and *fullName* in classes B and C are correctly matched with field *fullName* in class A, and all applied refactorings detected correctly.

$$\text{relationshipSimilarity} = \frac{\text{intersection}(\text{callers}(e_1), \text{callers}(e_2))}{callers(e_1)} \tag{1}$$

It is worth mentioning that for the *invokers* method (Line 15), this scenario is not established and all methods and fields called by the methods under investigation are examined. In fact, it is assumed that if the bodies of the two methods are different, they are likely two different methods, and pull up and push down refactorings do not make sense for two different methods.

IV. EVALUATION

To evaluate the feasibility of our approach in the practical environment, we conduct a series of experiments on Java and C++ applications. We also compare our tool, RefDetect with RMiner [3] which is the state-of-the-art tool for detecting refactorings in Java applications. In this section, we first present our research questions and validation methodology followed by evaluation setup. We then evaluate the quality of the approach based on precision, recall, and f-score. Time performance and memory consumption are also evaluated. The full results of our experiments can be found in the following link [34].

A. RESEARCH QUESTIONS

We defined three main research questions to measure the correctness, completeness, and performance of the implemented tool when applied to a real-world scenario. More specifically, our experiment addresses the following research questions:

- **RQ1:** To what extent can the proposed approach **correctly** detect the refactorings applied between two successive versions of a program?
- **RQ2:** To what extent can the proposed approach **completely** detect the all refactorings applied between two successive versions of a program?
- **RQ3:** How does our approach perform in terms of execution time and memory consumption?

B. EVALUATION SETUP

In this validation, we have focused on Java and C++ applications. To address the different research questions in Java applications, we used a dataset [35] which was initially

[16] The algorithm does not use any *lexical* or *semantic* similarity technique to determine how "close" are the name of entities. We use `equals()` method which tells that the input names are completely identical or not (see `haveSimilarName` function in Algorithm 2).

[17] As discussed, the matching algorithm allows one-to-many and many-to-one matching respectively for *PushDown* and *PullUp* refactoring types.

proposed by Silva *et al.* [25], and later expanded by Tsantalis *et al.* [3]. The dataset contains more than 7,000 true refactoring instances for 40 different refactoring types found in 536 commits from 185 open-source Java projects hosted on GitHub [3]. More importantly, all true refactoring instances are validated with multiple tools and by multiple experts [3], [11], [25], so we are confident that this dataset forms a suitable basis for evaluating RefDetect.

We used 514 of 536 commits from 185 open-source projects presented in the dataset. The reason that some commits were excluded is that some projects were not publicly available in GitHub anymore, and in a small number of cases, the employed Java source code analyser (Spoon) could not successfully parse source files in the input commit. We also restricted the dataset to the 27 refactoring types shown in Table 1. Specifically, we exclude some low-level refactoring types such as those related to variables (e.g. Rename/Inline Variable, etc.) and those related to annotations (e.g. Add/Remove Parameter, Class Annotation, etc.) which are not supported by our tool, and are in any case less interesting refactorings.

We compare our tool with the current state-of-the-art refactoring detection tool, RMiner [3] in terms of precision, recall and performance (execution time and memory consumption). To measure these metrics, we run both tools under the same environment and for each commit we analyse the detected refactorings. The precision and recall are measured using these standard formulae:

$$\text{precision} = \frac{\text{\# of correct refactorings}}{\text{\# of recommended refactorings}} \tag{2}$$

$$\text{recall} = \frac{\text{\# of correct refactorings}}{\text{\# of true refactorings in the dataset}} \tag{3}$$

We also used f-score as a combination of recall and precision to describe the effectiveness of each tool, where a value closer to 1 indicates a better performance.

$$\text{f-score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{4}$$

To address the various research questions in C++ applications, we conducted experiments using four C++ applications. However, the refactored program is created different than what has been described with Java applications. In fact, all applications used in the experiment contain some weaknesses in their design (known as design or code smells), and can be improved using refactoring operations. In each experiment, the application under investigation was refactored by two Masters students, and then we used our refactoring detection tool to identify refactorings applied by participants in each application. A detailed description of experiments done with C++ applications is provided in Section IV-D.

All of our experiments with Java and C++ applications are conducted on a computer with an Intel Core i5-8250U with 12GB of DDR3 memory, and a 5400 RPM HDD, running Windows 10 64-bit OS, Eclipse JDT 4.18, CLion 2021.1.1 and Java SE 15.0.1 x64. In the coming sections we first cover the experiments performed with Java applications and compare our tool with RMiner and then move on to experiments done with C++ applications.

C. RESULTS FOR JAVA APPLICATIONS

This section reports results for Java applications regarding the research questions presented in Section IV-A. The results are based on RefDetect 1.0 and RMiner 2.0.1.

1) DATASET CONSTRUCTION

As mentioned, in the experiments performed with Java applications, we compare our tool with RMiner based on the dataset [35] provided by Tsantalis *et al.* [3]. We analyse the detected refactorings as follows: If a refactoring is detected by both tools and is reported as true or false positive instance by Tsantalis *et al.* [3], we accept that as it is as determined in the dataset. However, if a refactoring is detected by one tool, but not by the other, we manually analyse the refactoring to know it is a true or false positive instance. This manual validation is done by the first and third author of this paper. It is important to note that as we are not the developers of Java applications used in the experiments, so it is possible that we made a mistake in categorizing the detected refactorings as false or true positive instances. However, in situations that RefDetect or RMiner cannot detect a correct refactoring or detect a refactoring incorrectly, where possible we simulated the situation using a simple test case and made this available on the project web page [34]. This allows other researchers to become aware of deficiencies in both these tools using some practical examples, and helps to develop better refactoring detection tools in the future.

Based on this procedure, 796 refactoring instances were detected only by RefDetect, while 548 cases were detected only by RMiner. Manual inspection revealed that among these refactorings, 85% in RefDetect and 90% in RMiner were detected correctly. The first impression from these numbers is that while both tools have some commonality, each covers some aspects not covered by the other tool.

We also extend the dataset used in this paper by including refactorings correctly detected by RefDiff 0.1.1 [9] or RefDiff 2.0 [11], but not detected by either RefDetect or RMiner. These refactorings are extracted from the list of refactorings reported by Tsantalis *et al.* [3]. Overall, **91** refactorings as depicted in Table 4, are added to the total number of *true refactoring instances*.

2) RESULTS FOR RQ1 & RQ2 IN JAVA APPLICATIONS

Table 5 presents a comparison of precision and recall in Java applications between RefDetect and RMiner based on the dataset described in the previous section. In total, data related to 27 refactoring types categorized into three groups according to their scope as shown in Table 1 were collected. In total, 5,058 refactorings included in the dataset, where

TABLE 4. True refactoring instances detected by RefDiff 0.1.1 [9] or RefDiff 2.0 [11], but not detected by either RefDetect or RMiner.

Refactoring Type	#TP	Refactoring Type	#TP
Extract & Move Method	43	Move Class	3
Rename Method	20	Move & Inline Method	2
Extract Method	13	Move & Rename Method	1
Move Method	4	Push Down Method	1
Inline Method	3	Rename Class	1

RefDetect was slightly better than RMiner in terms of f-score (87.3% vs. 86%). In fact, while RefDetect achieved a better recall than RMiner (84.5% vs. 78.9%), RMiner achieved a better precision (98.5% vs. 91.2%).

A detailed investigation reveals that while results for both tools in the majority of investigated refactoring types are very close and promising, but for some refactoring types (such as *Move and Rename Field*, *Inline Method*, or *Extract Class*), there are distinct differences between two tools. In addition, both tools have lower recall than precision, meaning that they miss some refactorings in comparison to the total number of true refactoring instances included in the dataset. In the next section, we investigate these differences in more detail and also present the reasons for incorrect and missed refactorings in both tools.

3) DETAILED REVIEW OF RESULTS

This section is divided into three subsections according to scope of detected refactoring types namely field, method and class-level refactoring types.

a: FIELD-LEVEL REFACTORING TYPES

We classified five refactoring types applicable to fields as field-level refactoring types as shown in Table 6. In total, 468 true field-level refactoring instances were present in the dataset. While RefDetect performs slightly better than RMiner in *PushDown Field* refactoring (f-score: 96% vs. 93.9%), RMiner outperforms RefDetect in the rest of field-level refactoring types (f-score: 89.3% vs. 82.1%). The worst case in both tools occurred for the *Move & Rename Field* refactoring type. However, a notable low precision (41.7%) and recall (50%) was recorded for this refactoring type in RefDetect.

b: DETAILED ANALYSIS FOR FIELD-LEVEL REFACTORINGS

A detailed investigation reveals that there are two factors that negatively affect the accuracy of RefDetect in detecting field-level refactoring types. We first discuss the main reason for false positive instances, and then describe a scenario that was the cause of half of false negative cases in RefDetect.

False positive cases: in total, 24 field-level refactorings are detected incorrectly by RefDetect. Our investigation shows that in the majority of these cases, the original field is deleted from the program and the target field is actually a newly created one (both through non-refactoring changes). The same scenario also happens for methods which use these fields. In fact, majority of methods that use the original field are deleted from the program, and majority of methods that use the new field are themselves new in the program. However, in all these cases, there were still few methods use both fields. As discussed, the similarity of two fields is measured based on similarity of their names as well as similarity of methods that use these fields (Lines 10 to 13 in Algorithm 2). The name of fields are different in majority of false positive cases detected by RefDetect (20 of 24), and therefore it has negative effect on similarity value. On the other hand, as described in section III-C1, *methods which are identified by the matching algorithm as deleted or added in the program are ignored in the relationship similarity measurement process*. Therefore, few methods which use both fields, result a high relationship similarity value between two fields. Consequently, the matching algorithm finds these fields as being the same, and detects an incorrect refactoring.

False negative cases: we find a scenario that was the cause of almost half of the false negative cases in RefDetect for field-level refactoring types. In all these false negative cases, the refactored field is called by methods mostly different than those called it in the original program. As a result, the matching algorithm finds a low relationship similarity between two fields and consequently cannot detect the applied refactoring. Note that, contrary to the false positive cases, the methods that call the field exist in both the original and refactored programs, and are not deleted from the original program or added to the refactored program.

RMiner, on the other hand, is more robust to aforementioned conditions which negatively affect the accuracy of RefDetect. This mainly happens as RMiner uses a completely different strategy for matching fields. In RMiner, all abstract syntax tree nodes including fields are matched in three rounds. "*In the first round, the nodes with identical string representation and nesting depth are matched. In the second round, the nodes with identical string representation regardless of their nesting depth are matched, and in the last round, the nodes that become identical after replacing the AST nodes being different between the two nodes are matched*" [3]. This strategy helps RMiner to be robust to changes affect the field relationships. For instance, RMiner correctly detects 40 *Move Field* refactorings not detected by RefDetect. In all these refactorings, the signature of the fields (including their names and types) are not changed, and RMiner correctly matches them based on their signatures. However, RefDetect cannot match correct fields as their relationships through non-refactoring activities have significantly changed.

The way RMiner matches fields also has its drawbacks. For example, in the experiments done in this paper, we found few cases that two fields with similar name and type, but different functionality were incorrectly matched by RMiner. We simulate this scenario using a simple test case observed in

TABLE 5. Precision, recall and f-score results per refactoring type.

		RefDetect 1.0			RMiner 2.0.1		
Refactoring Type	**#TP**	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Field-Level Refactorings	468	85.5	79.5	82.1	98.2	83.4	89.3
Class-Level Refactorings	1253	94	91	92.1	98.1	81.1	86.9
Method-Level Refactorings	3337	94.1	83	87.7	99.1	72.2	81.7
All Refactoring Types	**5058**	**91.2**	**84.5**	**87.3**	**98.5**	**78.9**	**86**

TABLE 6. Precision, recall and f-score results per field-level refactoring type.

		RefDetect 1.0			RMiner 2.0.1		
Refactoring Type	**#TP**	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Rename Field	115	88.5	87	87.7	97.8	79.8	87.9
Move Field	191	97.3	78.3	86.8	93.2	96.5	94.8
Push Down Field	26	100	92.3	96	100	88.5	93.9
Pull Up Field	126	100	89.7	94.6	100	96.8	98.4
Move & Rename Field	10	41.7	50	45.5	100	55.6	71.5
Field-Level Refactorings	**468**	**85.5**	**79.5**	**82.1**	**98.2**	**83.4**	**89.3**

Aeron-4b76, and hive-abe6 applications. In this test case, an existing field is deleted from the program and a new field with **similar name** and **type** is created in another class in the program. *However, the new field is created for a different functionality and it is used by completely different methods*. RMiner detects these two fields as being the same and incorrectly detects a *Move Field* refactoring. Obviously, the problem happens as methods which call the fields are not considered during fields matching process, and fields are matched based on their names and types.

In summary, non-refactoring activities which result significant changes in field relationships can negatively affect both precision and recall of RefDetect. The matching strategy used by RMiner allows it to cope with non-refactoring changes, but it has its drawback, and ignores field relationships in the matching process.[18]

c: CLASS-LEVEL REFACTORING TYPES

We classified seven refactoring types applicable to classes as class-level refactoring types as shown in Table 7. In total, 1,253 true class-level refactoring instances appeared in the dataset. As shown in Table 7, in terms of f-score, RefDetect performs better than RMiner overall (92.1% vs. 86.9%). RMiner achieved a better precision than RefDetect (98.1% vs. 94%), while a better recall is achieved by RefDetect (91% vs. 81.1%).

In terms of number of applied refactorings, *Move Class* refactoring with 1,049 true instances is the most applied refactoring (84% of class-level refactorings). It is worth mentioning that almost half of these refactorings occurred in only two applications. In fact, 298 and 193 *Move class* refactorings occurred in android-c976, and docx4j-e299 applications respectively. In these applications, a root package is renamed, and classes within this package are identified as moved class. It was an easy refactoring for both RefDetect and RMiner as minor changes were applied in the moved classes, and in most cases the moved classes were exactly the same as their corresponding classes in the original program.

d: DETAILED ANALYSIS FOR CLASS-LEVEL REFACTORINGS

As shown in Table 7, in terms of precision, *Extract Class* refactoring type has the weakest results in both tools. The lowest precision in both tools for this refactoring type is mainly due to some incorrectly detected *Move Method* refactorings (see rules defined in Table 2). Obviously, improving the technique used to match methods and fields can have positive effect on class-level refactoring types. However, an important note which should be mentioned is the way that *Extract Class* refactoring is defined in both RefDetect and RMiner. Both tools defined this refactoring as follows: *Extract Class refactoring involves creating a new class and moving **at least two** fields and/or methods from a class to*

[18] We also observed that RMiner is not capable of detecting refactorings applied to fields defined in the enum, a case that happened in facebook-android-sdk-19d, and giraph-03ad applications. A close investigation also reveals that 9 out of 16 false positive field related refactorings detected by RMiner, were caused by not detected *Rename Class* refactorings.

TABLE 7. Precision, recall and f-score results per class-level refactoring type.

		RefDetect 1.0			RMiner 2.0.1		
Refactoring Type	**#TP**	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Rename Class	50	93.6	89.8	91.7	100	90	94.7
Move Class	1049	99.7	98.5	99.1	99.8	98.6	99.2
Extract Superclass	31	100	93.5	96.6	100	100	100
Extract Subclass	4	100	75	85.7	100	75	85.7
Extract Class	73	81.6	97.3	88.8	91.7	30.1	45.3
Extract Interface	24	100	91.7	95.7	100	87.5	93.3
Move & Rename Class	22	83.3	90.9	86.9	95	86.4	90.5
Class-Level Refactorings	**1253**	**94**	**91**	**92.1**	**98.1**	**81.1**	**86.9**

the newly created class. However, this definition is a bit different than one defined by Fowler [33], where the fields and methods *responsible for a relevant functionality* placed in the new class. Obviously, the accuracy of *Extract Class* refactoring can be improved if the connection between methods and fields moved to the new class are also considered when deciding about this refactoring type.

RMiner also has the lowest recall for *Extract Class* refactoring (30.1%). We observer that 80% of *Extract Class* refactorings not detected by RMiner occurred in java-algorithms-implementation-ab98. In all these cases, an existing method is changed to a new inner class with a similar name, and a field is also moved to this newly created class. Fig. 9 depicted an instance extracted from java-algorithms-implementation-ab98. As shown, method *testTrie* is changed to an inner class with similar name, and field *trie* is also moved to this class. Method *run*, and *getName* are both newly created methods.

On the other hand, in RefDetect, one of weakest results in terms of recall is achieved for the *Rename Class* refactoring.[19] In our investigation, we noticed that the *Rename* and *Move Class* refactorings, being the only refactorings dependent on class similarity thresholds, are very sensitive to *non-refactoring activities* that change the number of methods and fields defined in classes. In fact, if more than 30% of methods and fields defined in a class are deleted or added as new elements to the class, then the similarity of the class in the original and refactored programs will be less than defined threshold and classes will not be detected as being the same (see Table 3). In fact, it was the main reason for the majority of *Move/Rename Class* refactorings not detected by RefDetect. It is important to recall that when the similarity between two classes is measured, the fields and methods that have been removed from the class in the original program or added to the class in the refactored program, both using non-refactoring changes, are considered as differences between two classes. Otherwise, the algorithm will regards a deleted class, and a completely new class as the same class. However, note that changes resulting from refactorings (e.g., if a method is moved from the original class to another class) are handled correctly by the algorithm, and they are referred to as similarities between the two classes.

[19] We ignore *Extract Subclass* refactoring as it only occurred four times.

```
- private static final Trie<String> trie = new Trie<String>();
- private static boolean testTrie() {
+ private static class testTrie extends Testable {
+     Trie<String> trie = new Trie<String>();
      String name = "Trie <String>";
                          trieCollection = trie.toCollection();

-     if (!testJavaCollection(trieCollection,String.class,name))
-         return false;
-     return true;
+     public String getName() {
+         return name;
+     }

+     public boolean run(Integer[] unsorted, Integer[] sorted,
                                                  String input) {
+         this.input = input;
+         if (!testJavaCollection(trieCollection,String.class,name,
+                                                 unsorted,sorted, input))
+           return false;
+         return true;
+     }
+
  }
```

FIGURE 9. Illustrative diff of one *Extract class* refactoring taken from java-algorithms-implementation-ab98 detected by RefDetect, but not detected by RMiner.

e: METHOD-LEVEL REFACTORING TYPES

We classified eleven refactoring types applicable to methods as method-level refactoring types as shown in Table 8. In total, there are 3,337 occurrences of method-level refactorings in the dataset. As shown in Table 8, in terms of f-score, RefDetect outperforms RMiner in detecting method-level refactorings (87.7% vs. 81.7%). While both tools exhibit a strong level of precision (RefDetect: 94.1%

TABLE 8. Precision, recall and f-score results per method-level refactoring type.

		RefDetect 1.0			RMiner 2.0.1		
Refactoring Type	**#TP**	**Precision**	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Rename Method	356	95.7	74.9	84	97.4	76.4	85.6
Move Method	198	91.8	91.8	91.8	98.8	90	94.2
Push Down Method	36	100	66.7	80	100	91.2	95.4
Pull Up Method	285	99.6	90	94.6	99.6	96.8	98.2
Extract Method	707	98.4	86.7	92.2	98.8	90.8	94.6
Inline Method	180	98.2	91.1	94.5	98	54.4	70
Change Method Parameters	1272	99.2	94.8	97	99	88.9	93.7
Move & Rename Method	48	84.3	91.5	87.8	100	54.3	70.4
Extract & Move Method	190	92.4	57.4	70.8	98.6	39	55.9
Move & Inline Method	32	82.1	71.9	76.7	100	40	57.1
Move & Change Method Parameters	33	93.9	96.9	95.4	100	71.9	83.7
Method-Level Refactorings	**3337**	**94.1**	**83**	**87.7**	**99.1**	**72.2**	**81.7**

and RMiner: 99.1%), the results in terms of recall show that both tools need some improvement (RefDetect: 83% and RMiner: 72.2%). In fact, each tool makes unique observations that are not evident to the other tool.

As discussed in Section III-E, when measuring the similarity of two methods, apart from comparing their signature, their similarity in terms of *methods that call them* as well as *methods and fields invoked/accessed by them* are also measured (Lines 13 and 15 in Algorithm 2). It is also worth mentioning that when a method is converted to a sequence of characters, we only extract information about methods and fields invoked/accessed by the method and we do not extract any information about programming instructions (e.g. `if`, `while` statements etc.), literals, comments, etc. used inside the method body. It is completely different than technique used by RMiner which extracts all information inside the method body. In the following, we will discuss the strengths and weaknesses of matching techniques employed by RefDetect and RMiner using examples observed in the experiments.

f: DETAILED ANALYSIS FOR METHOD-LEVEL REFACTORINGS

Our careful examination of refactorings not detected by RefDetect reveals that literals and comments used in the method body can be highly effective in finding method-level refactorings. For example, Fig. 10 shows a *Rename Method* refactoring that occurred in drools-1bf287. This refactoring is not detected by RefDetect as the method has a weak relationship with other fields and methods in the program. In fact, no method or field is invoked/accessed by the method, and the method is itself only called by one another method in the program. However, RMiner successfully detects this refactoring as the bodies of the methods (`return false;`) are a perfect match.

```
-   public boolean requiresImmediateFlushingIfNotFiring() {
+   public boolean requiresImmediateFlushing() {
        return false;
    }
```

FIGURE 10. Illustrative diff of a *Rename Method* refactoring taken from drools-1bf287 detected by RMiner, but not detected by RefDetect.

As another example, Fig. 11 illustrates a *Rename Method* refactoring occurred in mockito-7f20e6. In this case, while these methods are not called in the program, their body in terms of their relationships with other entities in the program are completely different from each other. Therefore, RefDetect cannot detect these methods as being the same. However, string literal (`<Capture argument>`) used in the method body helps RMiner to successfully detect this refactoring.

```
-   public void describeTo(Description description) {
-       description.appendText("<Capturing argument>");
+   public String describe() {
+       return "<Capturing argument>";
    }
```

FIGURE 11. Illustrative diff of a *Rename Method* refactoring taken from mockito-7f20e6 detected by RMiner, but not detected by RefDetect.

While literals used in the method body can help RMiner to find the right matches, our experiments show that changes in the literals can negatively affect the accuracy of RMiner. As an example, Fig. 12 shows a case observed in bitcoinj-7744, where boolean literals used in methods help RMiner to detect the right two *Move Method* refactorings. However, when even a boolean literal used in one of these methods is

changed, RMiner is no longer capable of detecting the right refactorings, and misidentified the applied refactorings.

A detailed analysis also reveal that refactoring and non-refactoring changes applied in the method body can negatively affect the accuracy of matching algorithm in both RefDetect and RMiner. Each tool uses a completely different technique to neutralize these destructive effects. In the following, we discuss advantages and disadvantages of each technique using cases observed in the experiments.

```
- public void testTransactionsLazyRetain() throws Exception {
-  testTransaction(MainNetParams.get(), txMessage, false, true, true);
-  testTransaction(unitTestParams, tx1BytesWithHeader, false, true, true);
-  testTransaction(unitTestParams, tx2BytesWithHeader, false, true, true);
+ public void testTransactionsRetain() throws Exception {
+  testTransaction(MainNetParams.get(), txMessage, false, true);
+  testTransaction(unitTestParams, tx1BytesWithHeader, false, true);
+  testTransaction(unitTestParams, tx2BytesWithHeader, false, true);
  }

- public void testTransactionsLazyNoRetain() throws Exception {
-  testTransaction(MainNetParams.get(), txMessage, false, true, false);
-  testTransaction(unitTestParams, tx1BytesWithHeader, false, true, false);
-  testTransaction(unitTestParams, tx2BytesWithHeader, false, true, false);
+ public void testTransactionsNoRetain() throws Exception {
+  testTransaction(MainNetParams.get(), txMessage, false, false);
+  testTransaction(unitTestParams, tx1BytesWithHeader, false, false);
+  testTransaction(unitTestParams, tx2BytesWithHeader, false, false);
  }
```

FIGURE 12. Illustrative diff of two *Rename Method* refactorings taken from bitcoinj-7744 detected by RMiner. We changed the first `true` literal in method *testTransactionsRetain()* to `false` and RMiner misidentified the refactorings.

As discussed in section III-C1, RefDetect uses a two-step conservative approach to match entities different in the original and refactored programs, where changes identified in the first round provide extra information that enables RefDetect to match entities more accurately in the second round. However, the implemented approach has a minor drawback as follows: *In a case that a method calls* ***an existing entity*** *through a new call relationship, or a call to* ***an existing entity*** *is deleted from a method body, then these changes are counted as differences between two methods while they are non-refactoring changes and should be ignored by the algorithm*. This drawback caused some of false negative cases in method-level refactorings (especially in *Extract Method* refactoring type) and field-level refactoring types (as discussed before).

As an example, Fig. 13 shows an *Extract and Move Method* refactoring as a composite refactoring that is not detected by RefDetect. As illustrated, the extracted method (*resizeTempBlockMeta*) contains new calls to two existing methods (*reverseSpace*, and *error*) and one field (*LOG*). These call instructions do not exist in the instructions removed from the method *requestSpace*. Consequently, the algorithm finds a low similarity between call relationships that exist in the extracted method and those deleted from the original method, and cannot detect the applied *Extract and Move Method* refactorings.

RMiner, on the other hand, uses a different technique which helps it to correctly detect the composite refactoring depicted in Fig. 13. RMiner, in fact, employs a replacement technique and uses some predefined heuristics to cover refactoring and non-refactoring changes applied in the method body. The replacement function receives as input two statements, and replaces each node in the first statement with all possible nodes in the second statement until the statements become textually identical. Two entities with the smallest edit distance are then determined as matched ones [3]. RMiner does not replace AST nodes that cover the entire statement (e.g., method invocations, and class instance creations), and instead uses some predefined heuristics to finds their similarity [3]. As an example, two method invocations are identified as being the same if they have similar receiver and similar list of arguments.[20] The replacement technique and heuristics used by RMiner allow it to detect some refactorings not detected by RefDetect. However, there are some drawbacks assigned with this approach which are discussed through a number of examples.

```
public class TieredBlockStore implements BlockStore {
  public boolean requestSpace(long userId, long blockId, long moreBytes){
-   tempBlock.setBlockSize(tempBlock.getBlockSize() + moreBytes);
+   mMetaManager.resizeTempBlockMeta(tempBlock, tempBlock.getBlockSize()
+                                                    + moreBytes);
    return true;
  }
}

public class StorageDir {
+ public void resizeTempBlockMeta(TempBlockMeta tempBlockMeta, long newSize)
+  {
+    long oldSize = tempBlockMeta.getBlockSize();
+    tempBlockMeta.setBlockSize(newSize);
+    if (newSize > oldSize) {
+      reserveSpace(newSize - oldSize);
+    } else {
+      LOG.error("Shrinking block, not supported!");
+    }
+ }
}
```

FIGURE 13. Illustrative diff of one *Extract and Move Method* refactoring taken from Alluxio-ed96 detected by RMiner, but not detected by RefDetect.

As first example, Fig. 14 illustrates a *Rename Method* refactoring detected by RefDetect, but not detected by RMiner. RefDetect uses method invocations in the methods' bodies to detect that the two methods *foo* and *bar* are the same, but RMiner cannot detect the refactoring as it finds very little similarity between methods *foo* and *bar*. The problem occurs as the replacement function does not

[20]Using this heuristic, method invocation objA.foo(i); is equal to objA.bar(i); as they both have similar receiver (*objA*), and both methods *foo* and *bar* have similar list of arguments. Note that this heuristic helps RMiner to cover a case that method *foo* is renamed to *bar*.

```
public class A {
-  public boolean foo(List<Integer> array){
-      int i = 0;
-      while (i < array.size()) {
+  public boolean bar(List<Integer> array){
+      for (int i = 0; i < array.size(); i++) {
            if (array.get(i) < 10) return true;
-           i++;
       }
       return false;
   }
}
```

FIGURE 14. Illustrative diff of a *Rename Method* refactoring detected by RefDetect, but not detected by RMiner.

cover the replacement of the `while` instruction with `for`. Note that while from a programmer's point of view it is a common replacement and the programmer can easily detect the similarity of the two methods, *it is hard to define and covering all possible replacement types in the matching algorithm*.

As another example, Fig. 15 shows an *Inline Method* refactoring occurred in truth-1768. This refactoring happened 67 times in this application and it is the main reason for a low recall for the *Inline Method* refactoring type in RMiner. As mentioned, RMiner does not replace AST nodes that cover the entire statement such as a method invocation, and instead uses some heuristics to cover these cases. However, none of defined heuristics helps RMiner to detect that the instruction `asList(1, 2, 3)` is the same as the instruction `Arrays.asList(items)` defined in the method `iterable`. Therefore, it cannot detect the applied *Inline Method* refactoring.

```
import static java.util.Arrays.asList;
 public void iterableContainsItem() {
-  assertThat(iterable(1, 2, 3)).contains(1);
+  assertThat(asList(1, 2, 3)).contains(1);
 }
-private static Iterable<Object> iterable(Object...items){
-   return Arrays.asList(items);
- }
```

FIGURE 15. Illustrative diff of a *Inline Method* refactoring taken from truth-1768 detected by RefDetect, but not detected by RMiner. This refactoring happened 67 times in this application.

As discussed through the aforementioned examples, despite the commonalities between RefDetect and RMiner, there are considerable differences between them, and these differences in some cases made one tool superior to another. However, a very important note is that there are still details in the refactoring detection process that have not been seen by any of these two tools.[21]

As an example, Fig. 16 shows an *Extract & Move Method* refactoring occurred in application cassandra-ec52. This refactoring is only detected by RefDiff 0.1.1, and it is not detected by RefDetect, RMiner and even RefDiff 2.0 that is the latest version of RefDiff. As depicted, the extracted method (*geBackgroundCompactionTaskSubmitter*) contains around two times more instructions than those deleted from the original method. This results a similarity value lower than similarity threshold selected in RefDetect as well as RefDiff 2.0. RMiner also cannot detect this refactoring as it cannot find any smaller edit distance than the one that exists between the deleted instructions and the instructions that exist in the extracted method. However, RefDiff 0.1.1 detects this refactoring as it uses a set of thresholds calibrated for each refactoring type [9]. In this case, a very low threshold value allows RefDiff 0.1.1 to detect the refactorings correctly.[22]

g: ANSWER TO RQ1 AND RQ2

In our evaluation with 514 commits of 185 Java applications containing 5,508 real refactorings, RefDetect achieved an f-score slightly better than that achieved by RMiner (87.3% vs. 86%). RefDetect clearly outperformed RMiner in class and method based refactorings, achieving f-scores respectively of 92.1% vs. 86.9% for class-level refactorings, and 87.7% vs. 81.7% for method-level refactorings. However, while RefDetect worked better in terms of recall (RefDetect: 84.5%, RMiner: 78.9%), RMiner scored extremely well in terms of precision (RefDetect: 91.2%, RMiner: 98.5%). We observed that RefDetect is sensitive to non-refactoring activities that result significant changes in relationships of refactored entities. It was also observed that literals and comments used inside method bodies are other language-independent constructs that can be highly effective in finding method-level refactorings. As future work, we intend to improve the proposed approach by addressing these two issues.

4) EVALUATION ON PERFORMANCE IN JAVA APPLICATIONS (RQ3)

We explored the performance of our approach based on two criteria: run-time and memory consumption. For this purpose, we measured these two metrics in both RefDetect and RMiner for all 514 commits under investigation. We measured the time and memory used by each tool to parse the Java files changed between the examined commit and its parent one, and also time and memory required to detect changes between two commits and to categorize the detected changes as refactoring instances.

[21] A list of 91 refactorings included in the dataset, but not detected by either RefDetect or RMiner can be found at the following address [34].

[22] The threshold value for *Extract Method* is equal to 0.1, and it is 0.4 for *Move Method* refactoring type (see Table 3 in [9]).

```
  public class CassandraDaemon {
   protected void setup() {
-   Runnable runnable = new Runnable() {
-       public void run() {
-           for (Keyspace keyspaceName : Keyspace.all()){
-               for (ColumnFamilyStore cf : keyspaceName.getColumnFamilyStores()) {
-                   for (ColumnFamilyStore store : cf.concatWithIndexes())
-                       CompactionManager.instance.submitBackground(store);
-               }
-           }
-       }
-   };
-   StorageService.optionalTasks.schedule(runnable, 5 * 60, TimeUnit.SECONDS);
+   StorageService.optionalTasks.scheduleWithFixedDelay(ColumnFamilyStore.getBackgroundCompactionTaskSubmitter(), 5, 1, TimeUnit.MINUTES);
    }
  }

  public class ColumnFamilyStore implements ColumnFamilyStoreMBean {
+  public static Runnable getBackgroundCompactionTaskSubmitter(){
+      return new Runnable(){
+          public void run(){
+              List<ColumnFamilyStore> submitted = new ArrayList<>();
+              for (Keyspace keyspace : Keyspace.all())
+                  for (ColumnFamilyStore cfs : keyspace.getColumnFamilyStores())
+                      if (!CompactionManager.instance.submitBackground(cfs, false).isEmpty())
+                          submitted.add(cfs);
+              while (!submitted.isEmpty() && CompactionManager.instance.getActiveCompactions() < CompactionManager.instance.getMaximumCompactorThreads()){
+                  List<ColumnFamilyStore> submitMore = ImmutableList.copyOf(submitted);
+                  submitted.clear();
+                  for (ColumnFamilyStore cfs : submitMore)
+                      if (!CompactionManager.instance.submitBackground(cfs, false).isEmpty())
+                          submitted.add(cfs);
+              }
+          }
+      };
+  }
  }
```

FIGURE 16. Illustrative diff of an *Extract & Move Method* refactoring taken from cassandra-ec52 detected by RefDiff 0.1.1, but not detected by either RefDetect, RMiner or RefDiff 2.0.

Similar to RMiner, we employed the JGit API to extract contents of files changed between two commits into a Git repository. Obviously, this technique significantly improves the efficiency of both tools in terms of the run-time and memory consumption. While both tools are capable of detecting refactorings in a remote repository without cloning the repository locally, to have a fair comparison and remove the impact of Internet speed, all repositories under investigation are first cloned locally, and then files changed in two successive commits were determined using the JGit API.

a: COMPARISON OF EXECUTION TIME

Execution time is determined as the time difference between beginning and end of processing as measured using calls to the *System.nanoTime()* Java method. Fig. 17 shows a violin plot of the distribution of the execution time for both RefDetect and RMiner. Table 9, on the other hand, shows details not evident in Fig. 17.

TABLE 9. Execution time in detail for all 514 commits.

	Min. (s)	Max. (s)	Avg. (s)	Median (s)	Total (min)
RefDetect	0.06	78	1.5	0.4	12.50
RMiner	0.002	58	1.3	0.1	10.88

As shown in Table 9, RMiner is on average 1.2 times faster than RefDetect (1.3 vs. 1.5). However, RefDetect is still time efficient as 76% of commits are processed in less than 1 second. In total, it takes 10.88 and 12.50 minutes for RMiner and RefDetect respectively to process all 514 commits.

This is a clear message that both tools are efficient in terms of execution time. However, to gain further insight into possible performance bottlenecks of the two tools, we subjected to greater scrutiny applications with an execution time higher than 20 seconds.

In RefDetect for all four cases,[23] the performance bottleneck occurred during parsing of the input source code and converting the code to a string representation (Step 1.1 and 1.2 in Fig. 1), and in all these cases the refactoring detection algorithm (Step 2 and 3 in Fig. 1) was fast. This pattern is observed for all 514 considered commits, where the time devoted to source code information extraction was higher than time taken by the refactoring detection algorithm. Therefore, finding the fastest parser for the programming language under investigation is the first priority in improving the performance of RefDetect.

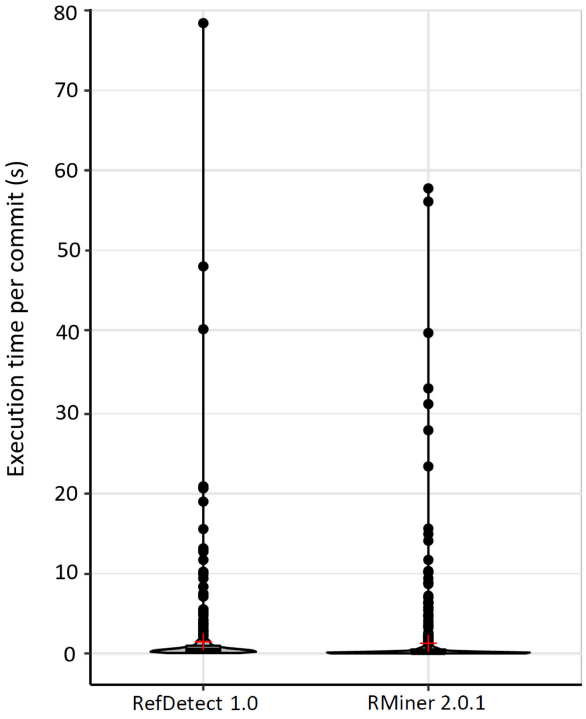


FIGURE 17. Violin plot of execution time per commit.

We also investigated seven applications[24] that resulted in an execution time longer than 20 seconds in RMiner. The growth in the execution time appears to be caused mainly by the number of **replacement operations** performed by the RMiner matching algorithm, which depends heavily on the number of statements changed between two commits. In contrast, the RefDetect matching algorithm is independent of the number of statements changed between two commits. As a result, RefDetect's execution time for all seven applications for which RMiner's execution time exceeds 20 seconds, is only 17 seconds in total.

b: COMPARISON OF MEMORY CONSUMPTION

Memory consumption is defined as the amount of memory used by each tool during the refactoring detection process. To measure the used memory, we employed the *Runtime.getRuntime().freeMemory()* as well as *Runtime.getRuntime().totalMemory()* Java methods. We estimated memory consumption in a crude fashion by simply subtracting the memory used at the start of the process from the memory in use when the process terminates.

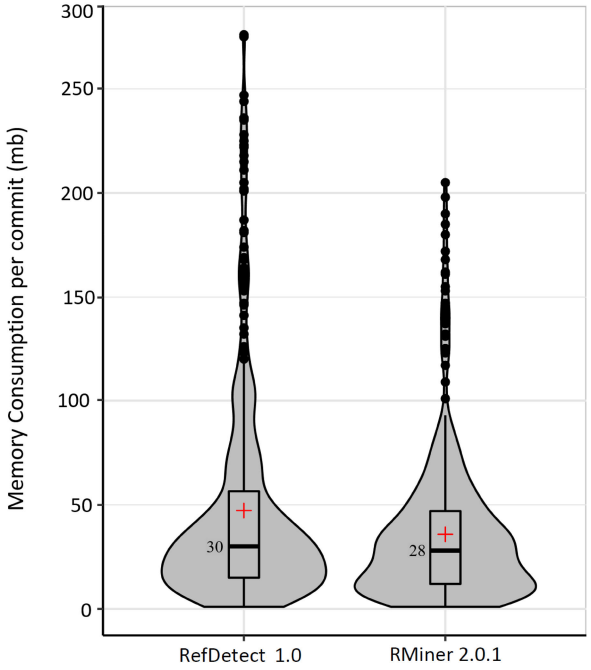


FIGURE 18. Violin plot of memory consumption per commit in megabytes.

Fig. 18 shows a violin plot of the memory used per commit in terms of megabytes for both RefDetect and RMiner. In terms of the median value, both tools require nearly the same memory (30 vs. 28), but on average, RMiner is more efficient, and it used almost 20% less memory than RefDetect (47 vs. 36).

c: ANSWER TO RQ3

According to the results illustrated in Fig. 17, and 18, RefDetect is around 20% less efficient than RMiner in terms of execution time and memory consumption. However, RefDetect is still efficient and its performance is perfectly acceptable from a pragmatic point of view.

D. RESULTS FOR C++ APPLICATIONS

This section reports results for detecting refactorings in C++ applications. To the best of our knowledge, there is no public refactoring dataset for C++ applications like the one available for Java applications [35]. Recently, Silva *et al.* [11]

[23] byte-buddy-372f, clojure-309c, android-frameworks-9103, and docx4j-e299

[24] java-algorithms-ab9, cordova-plugin-51f, netty-d31, hazelcast-30c, jbpm-381, jfinal-881, and cassandra-446.

provided a refactoring dataset that contains 90 true refactoring instances extracted from 20 C applications (on average around five refactorings in each application). However, apart from its small scale, the refactorings are extracted from C applications and therefore do not contain any class-level refactoring types (such extract class, etc.). In addition, it does not contain any field-level refactoring types (such as move field, etc.) as the provided tool: RefDiff 2.0 [11] does not support any field-level refactoring types. Overall, it only contains 6 out of the 27 refactoring types supported in RefDetect. Note that were we to extend this dataset, then as we are not the original developers of the applications, it is likely that we would make errors in classifying the detected refactorings as true or false positive instances due to experimenter bias [2]. In addition, there is the possibility that we would miss some true refactoring instances because of an algorithm design flaw, inappropriate threshold values etc. (i.e. biased oracle) [2]. Therefore, we follow another route to evaluate RefDetect with C++ applications.

We conducted experiments using four C++ applications illustrated in Table 10. All applications contain some weaknesses in their design (known as design or code smells), and can be improved using refactoring operations. Each application is refactored independently by two Master's students who previously completed a software refactoring course taught by the first author of this paper. All participants involved in the experiment were familiar with applications under investigations and worked before with CLion IDE which was used to refactor the applications. Overall, 10 students voluntarily participated in this experiment.

TABLE 10. C++ applications used in this study.

No.	Application	#Classes	#Methods	#Fields
1	Shopping Mall	16	93	33
2	ATM Simulation	5	31	9
3	Animal Simulation	20	53	21
4	Bank Simulation	22	98	88

We asked participants to first analyse the code assigned to them, and then improve the program structure using refactoring operations. Participants were free to do the refactorings manually or using refactorings provided by the CLion IDE. We then selected from the two refactored programs the one which contains more complete and precise refactorings. To simulate non-refactoring changes applied in a real scenario, we manually injected some non-refactoring changes into all refactored applications which resulted in four new refactored applications. For instance, we manually injected some new instructions to an extracted methods, or define some new fields and methods to a renamed class. We also created another group of applications as follows: we treat the versions refactored by participants also containing non-refactoring changes as the new original programs and select the real original programs as the refactored ones.

The results for experiments performed with C++ applications are presented in Table 11. As shown, overall, 305 refactorings were applied to applications under investigation, where RefDetect achieved on average 96.1% precision and 94.1% recall. These results show the feasibility of the proposed approach for detecting refactorings in programming languages other than Java. Not surprisingly, all incorrectly detected refactorings as well as missed refactorings were the result of non-refactoring changes injected into the applications.

TABLE 11. C++ precision and recall results.

Refactoring Type	#TP	Precision	Recall
Rename Class	16	100	87.5
Extract Superclass	6	100	100
Extract Subclass	6	100	100
Collapse Hierarchy	12	100	100
Extract Class	4	100	100
Inline Class	4	100	100
Rename Field	10	80	80
Move Field	6	75	100
Push Down Field	22	100	91
Pull Up Field	22	100	91
Move & Rename Field	4	100	100
Rename Method	17	100	88
Move Method	6	75	100
Push Down Method	65	100	93.8
Pull Up Method	65	100	93.8
Extract Method	14	100	85.7
Inline Method	14	100	85.7
Move & Rename Method	12	100	100
Total	**305**	**96.1**	**94.1**

In summary, our evaluation with C++ applications provides some evidence that our approach is indeed language independent and can easily be ported to work with another language. Java and C++ are closely related languages, and our approach only deals with the object-oriented constructs of these languages. We would anticipate that similar results would be observed using object-oriented code written in other related languages, e.g. C#. Applying the RefDetect approach to non-OO code, e.g. lambda expressions in Java or closures in C# is an area for exploration, as is applying it to non-OO languages such as C or Haskell.

A more complete analysis of how well RefDetect works for applications written in C++ and other languages is left for future work.

V. LIMITATIONS

Despite the encouraging results, some of the investigated scenarios indicated potential limitations. This section focuses on limitations of the approach itself, and sketch some possible extensions to address them.

A. MISSING CHANGES

As described in section III-B, the employed alignment algorithm, FOGSAA, decides about differences in the two input strings by comparing the type and name of entities included in the strings. However, since the algorithm does not consider the relationships between entities, the detected differences might not all be correct. As an example, consider changes applied in class *A* illustrated in Fig. 19a. In this example, Field *location* and method *getLocation()* are removed from class A, and field *address* and method *getAddress()* are renamed to *location* and *getLocation()* respectively. In this example, FOGSAA incorrectly identifies fields *location* as well as methods *getLocation()* in classes A and B as being the same, and recognizes field *address* and method *getAddress()* as the only differences between classes A and B. The result is that the refactoring detection algorithm cannot detect any of applied refactorings.

This problem can be solved by comparing relationships between entities in addition to their names and types. However, examining all entities' relationships significantly increases the execution time of FOGSAA. In addition, in the evaluations done in this paper, we observed few cases like one illustrated in Fig. 19a. It is worth mentioning that the relationships between different entities in the original and refactored programs are compared with each other by the matching algorithm (Step 2 in Algorithm 1). However, as only the relationships of different entities in two programs are compared, and a commit usually leaves most entities unchanged, this has only a minor effect on the speed of the algorithm.

RMiner is not also capable of detecting any of the refactorings applied in Fig. 19a. As mentioned in Section IV-B, RMiner matches entities in three rounds, where "*in the first round, the nodes with identical string representation and nesting depth are matched*" [3]. In this example, fields *location* and methods *getLocation()* are incorrectly matched in the first round, and it prevents RMiner from detecting the applied Rename refactorings.

B. LACK OF SEMANTIC INFORMATION SUPPORT

The entity matching function (presented as Algorithm 2) is currently based on structural information (type, name and relationships between entities). While the results of the experiments done in this paper show that this criterion is sufficient to correctly identify refactorings, there are cases that the algorithm is not able to correctly detect applied refactorings.

As an example, consider changes applied in classes in Fig. 19b. In this example, class *Reptile* and class *Fish* are respectively renamed to *Turtle*, and *Shark*. However, RefDetect cannot detect any of the applied *Rename Class* refactorings. This happens because each class in the original program completely matches with both classes in the refactored program. In fact, class *Reptile* is identified as 100% similar to classes *Turtle*, and *Shark*. The same also happens for class *Fish*. The refactorings are not detected as the matching algorithm is designed to prevent one-to-many matching between entities.

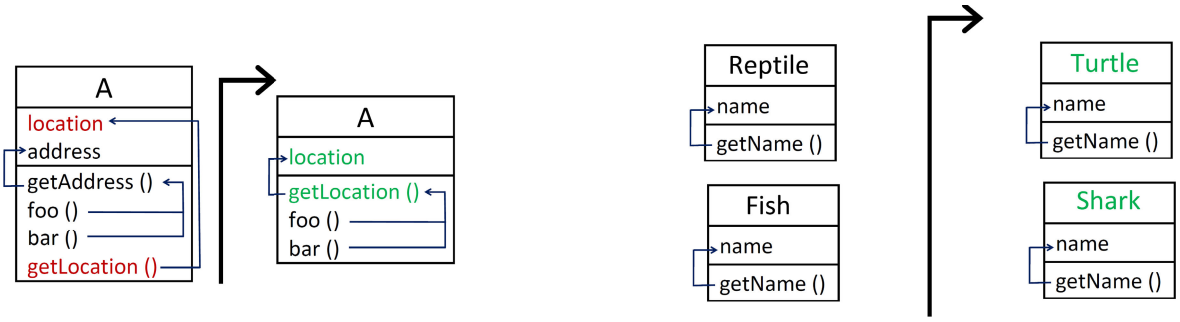
It is obvious that using both structural and semantic information when comparing entities can help to detect the correct matches. Two entities are semantically similar if they use similar words. For example, two classes are semantically similar if their names as well as names of their fields, methods, etc. have a high semantic similarity. The determination of the semantic characteristics can be handled using string comparison algorithms [36], through a lexical database of English such as WordNet,[25] or using well-known word embedding methods such as word2vec[26] or fastText.[27] In this example, semantic information can help to correctly identify relationship between *Fish* and *Shark* classes as well as *Reptile* and *Turtle* classes, and then the algorithm can correctly identify the applied refactorings. As future work, we intend to combine structural and semantic information to have a more precise correspondence between entities in the original and refactored programs during the entity matching process.

VI. THREATS TO VALIDITY

In any experimental study, some factors influence the findings and represent threats to validity. We discussed some limitation of our approach in the previous section, and in this section, we discuss other threats that can affect the validity of our experiments.

Regarding to the experiments done with Java applications, there are two important threats namely *bias in the dataset*, and *experimenter bias*. As discussed in Section IV-B, if a refactoring is detected by both RefDetect and RMiner, and it is reported as true or false positive instance by Tsantalis *et al.* [3], we accept that as it is as determined in the dataset. However, if a refactoring is detected by one tool, but not by another, we manually analysed the refactoring to determine it is a true or false positive one. While the refactorings included in the dataset are validated with multiple tools and by multiple experts [3], [11], [25], there were *very few cases* where we classified a refactoring that was identified as true one in the dataset as a false instance. On the other hand, 796 refactoring instances were only detected by our tool, where 85% of these refactorings were classified by us as true refactoring instances. However, we are aware that it is completely possible that we incorrectly classified a refactoring in our manual validation as we are not the developers of the applications under investigation (*experimenter bias*).

[25] https://wordnet.princeton.edu/
[26] https://code.google.com/archive/p/word2vec/
[27] https://fasttext.cc/



(a) Field *location* and method *getLocation()* are Removed from class A, and field *address* and method *getAddress()* are Renamed to *location* and *getLocation()* respectively.

(b) Class *Reptile* and class *Fish* are respectively Renamed to *Turtle*, and *Shark*.

FIGURE 19. Examples where the matching algorithm cannot match entities correctly.

Furthermore, new refactorings detected by our tool show that there can be still other refactoring instances not included in the dataset (*bias in the dataset*). As discussed in Section IV-B, there are still details in the refactoring detection process that have not been seen by any of RefDetect, and RMiner. Therefore, improvement in these tools, and even new tools in this direction may result lower recall value for our tool than one presented in this paper.

Regarding to the experiments done with C++ applications, there are some threats as follows. The first threat is its small scale. Overall, 305 refactorings were applied to eight applications (with each application being considered twice). It is also important to acknowledge that while we manually injected some non-refactoring changes to these applications, the changes might be meaningless, as our aim was to only create a refactored version containing non-refactoring changes. In addition, we are fully aware of the shortcomings of our tool in covering non-refactoring changes, and so it was better if non-refactoring changes were applied by a person not involved in developing RefDetect. Another threat to validity with the experiments done with C++ applications is that the participants took part in the experiment did not have the knowledge of an experienced software engineer. However, they all were Master's students who had previously studied a software refactoring course, and were familiar with applications under investigation. In summary, we acknowledge that our evaluation with C++ applications is not complete compared to the evaluation done with Java applications, and to draw stronger conclusions, further evaluation is needed. However, our main aim was to show the applicability of the proposed approach in detecting refactorings in other programming languages, and this was achieved.

Other threats to validity are related to values assigned to match, mismatch and gap penalty used in FOGSAA algorithm (see section III-B) as well as values assigned to similarity thresholds used by the matching algorithm (see Table 3). There is no doubt that to gain the best result, it is important to assign the proper values to these variables. However, as discussed by Tsantalis *et al.* [3], *finding a universal threshold value which works for all applications might be infeasible*. To moderate this threat, we experimentally calibrate values used by FOGSAA algorithm as well as similarity thresholds used by the matching algorithm, and also run the matching algorithm in two rounds with different similarity thresholds. However, as discussed in the evaluation section, the selected thresholds can still be further optimized for some refactoring types, especially when relationships between entities are significantly changed using non-refactoring changes.

VII. CONCLUSION AND FUTURE WORK

In this paper, we addressed the topic of identifying refactorings applied between two versions of a program. We proposed and implemented a novel refactoring detection algorithm that is not dependent on any programming languages. This is achieved through (i) representing the input programs as sequence of characters and abstracting away the specific details of each programming language, and (ii) relying only on entities' names and their relationships to match entities that have changed between the original and refactored programs. The viability of the implemented tool, called RefDetect, is demonstrated through its application to a number of Java and C++ applications. To the best of our knowledge, RefDetect is the first language-neutral refactoring detection tool to achieve a high level of accuracy in refactoring detection.

To evaluate the effectiveness of the proposed approach, we compare RefDetect with RMiner as the current state-of-the-art refactoring detection tool in Java. We conducted an experiment with a dataset containing 514 commits from 185 open source Java repositories, and including 5,058 true refactoring instances. The results show that while RMiner achieved a better precision (98.5% vs. 91.2%), RefDetect performed better in terms of recall (84.5% vs. 78.9%). Overall, in terms of f-score RefDetect was a bit better than RMiner (87.3% vs. 86). More specifically, while RMiner was good at detecting field-level refactoring types, RefDetect performed better in method- and class-level refactoring types. We also validated our approach on four C++ applications, where

RefDetect successfully detects 94% of 305 applied refactorings, and only 4% of detected refactorings were detected incorrectly.

We envisage future work taking place in a number of directions. Firstly we intend to improve the crucial entity matching process by combining structural and semantic information to create a more precise correspondence between entities in the original and the refactored programs. Secondly, while we have demonstrated language independence in this paper, we will use RefDetect to explore further refactoring detection in C++ programs. Finally, RefDetect is currently only concerned with refactoring detection, but it would also be useful to detect the order in which refactorings were applied in a commit. We plan to achieve this by simulating the refactoring effects on the string representation of the input program, without actually applying the detected refactorings to the program. This paves the way for RefDetect to form part of a larger framework that helps the developer to have a better understanding of the evolution of the program.

ACKNOWLEDGMENT

The authors would like to thank Dr. Nikolaos Tsantalis and his colleagues for their publicly available refactoring dataset, and their publicly available refactoring detection tool, which enhanced greatly our ability to evaluate our own tool.

REFERENCES

[1] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," Dept. Comput. Inf. Sci., Univ. Michigan, Dearborn, MI, USA, Tech. Rep., 2020. [Online]. Available: https://arxiv.org/abs/2007.02194

[2] 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.*, May 2018, pp. 483–494.

[3] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Trans. Softw. Eng.*, early access, Jul. 8, 2020, doi: 10.1109/TSE.2020.3007722.

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

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

[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proc. Eur. Conf. Object-Oriented Program. (ECOOP)*. Berlin, Germany: Springer, 2006, pp. 404–428.

[7] Z. Xing and E. Stroulia, "Differencing logical UML models," *J. Automated Softw. Eng.*, vol. 14, no. 2, pp. 215–259, 2007.

[8] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–10.

[9] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 269–279.

[10] R. Stevens, T. Molderez, and C. De Roover, "Querying distilled code changes to extract executable transformations," *Empirical Softw. Eng.*, vol. 24, no. 1, pp. 491–535, Feb. 2019.

[11] D. Silva, J. Silva, G. J. De Souza Santos, R. Terra, and M. T. O. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Trans. Softw. Eng.*, early access, Jan. 22, 2020, doi: 10.1109/TSE.2020.2968072.

[12] R. Krasniqi and J. Cleland-Huang, "Enhancing source code refactoring detection with explanations from commit messages," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2020, pp. 512–516.

[13] Q. D. Soetens, R. Robbes, and S. Demeyer, "Changes as first-class citizens: A research perspective on modern software tooling," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–38, Jun. 2017.

[14] A. Chakraborty and S. Bandyopadhyay, "FOGSAA: Fast optimal global sequence alignment algorithm," *Sci. Rep.*, vol. 3, no. 1, pp. 1–9, Dec. 2013.

[15] 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.

[16] 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.

[17] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. Compress. Complex. SEQUENCES*, Jun. 1997, pp. 21–29.

[18] 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 (MSR)*, 2011, pp. 53–62.

[19] B. Biegel and S. Diehl, "Highly configurable and extensible code clone detection," in *Proc. 17th Work. Conf. Reverse Eng.*, Oct. 2010, pp. 237–241.

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

[21] I. H. Moghadam and M. Ó. Cinnéide, "Automated refactoring using design differencing," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2012, pp. 43–52.

[22] 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.

[23] L. Tan and C. Bockisch, "A survey of refactoring detection tools," in *Proc. 6th Collaborative Workshop Evol. Maintenance Long-Living Syst. (EMLS)*, 2019, pp. 100–105.

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

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

[26] P. Langer, M. Wimmer, P. Brosch, M. Herrmannsdörfer, M. Seidl, K. Wieland, and G. Kappel, "A posteriori operation detection in evolving software models," *J. Syst. Softw.*, vol. 86, no. 2, pp. 551–566, Feb. 2013.

[27] F. Steimann, "Refactoring tools and their kin," in *Grand Timely Topics in Software Engineering* (Lecture Notes in Computer Science), vol. 10223, J. Cunha, J. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds. Cham, Switzerland: Springer, 2017, doi: 10.1007/978-3-319-60074-1_8.

[28] C. De Roover and R. Stevens, "Building development tools interactively using the EKEKO meta-programming library," in *Proc. Softw. Evol. Week IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 429–433.

[29] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Softw. Qual. J.*, vol. 21, no. 4, pp. 551–571, Dec. 2013.

[30] I. Hemati Moghadam, "Interactive software design improvement using metrics-driven, multi-level automated refactoring," Ph.D. dissertation, School Comput. Sci., Univ. College Dublin, Dublin, Ireland, 2014.

[31] I. Hemati Moghadam and M. Ó Cinnéide, "Resolving conflict and dependency in refactoring to a desired design," *e-Inform. Softw. Eng. J.*, vol. 9, no. 1, pp. 37–56, 2015.

[32] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[33] M. Fowler, *Refactoring: Improving Design Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[34] *RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment*. Accessed: Jun. 2021. [Online]. Available: https://sites.google.com/view/refdetect/home

[35] N. Tsantalis, A. Ketkar, and D. Dig. *Refactoring Oracle*. Accessed: Mar. 2021. [Online]. Available: http://refactoring.encs.concordia.ca/oracle/

[36] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proc. Workshop Inf. Integr. Web (IIWeb)*, 2003, pp. 73–78.

IMAN HEMATI MOGHADAM received the Ph.D. degree in software engineering from University College Dublin, Ireland, in 2014. He is currently an Assistant Professor with the Department of Computer Engineering, Vali-e-Asr University of Rafsanjan, Iran. Prior to this, he was a Research Associate with the Centre for Research on Evolution Search and Testing (CREST), Department of Computer Science, University College London, U.K. His primary research interests are software refactoring, search-based software engineering, and model-driven development.

MEL Ó CINNÉIDE received the Ph.D. degree in computer science from Trinity College Dublin, Dublin, Ireland, in 2001. He is currently an Associate Professor with the School of Computer Science, National University of Ireland, Dublin. His research interests include refactoring and design patterns. He is a member of the ACM and a chartered engineer.

FAEZEH ZAREPOUR received the B.S. degree in mathematics from the Vali-e-Asr University of Rafsanjan, Rafsanjan, Iran, in 2010, and the M.S. degree in software engineering from the Allameh Jafari Institute of Rafsanjan, Rafsanjan, in 2021. Her primary research interests are refactoring and software quality.

MOHAMAD AREF JAHANMIR is currently pursuing the B.S. degree in software engineering with the Vali-e-Asr University of Rafsanjan, Iran. His primary research interest is software refactoring.