RefDiff 2.0: A Multi-language Refactoring Detection Tool

Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente, *Member, IEEE*

Abstract—Identifying refactoring operations in source code changes is valuable to understand software evolution. Therefore, several tools have been proposed to automatically detect refactorings applied in a system by comparing source code between revisions. The availability of such infrastructure has enabled researchers to study refactoring practice in large scale, leading to important advances on refactoring knowledge. However, although a plethora of programming languages are used in practice, the vast majority of existing studies are restricted to the Java language due to limitations of the underlying tools. This fact poses an important threat to external validity. Thus, to overcome such limitation, in this paper we propose RefDiff 2.0, a multi-language refactoring detection tool. Our approach leverages techniques proposed in our previous work and introduces a novel refactoring detection algorithm that relies on the Code Structure Tree (CST), a simple yet powerful representation of the source code that abstracts away the specificities of particular programming languages. Despite its language-agnostic design, our evaluation shows that RefDiff's precision (96%) and recall (80%) are on par with state-of-the-art refactoring detection approaches specialized in the Java language. Our modular architecture also enables one to seamlessly extend RefDiff to support other languages via a plugin system. As a proof of this, we implemented plugins to support two other popular programming languages: JavaScript and C. Our evaluation in these languages reveals that precision and recall ranges from 88% to 91%. With these results, we envision RefDiff as a viable alternative for breaking the single-language barrier in refactoring research and in practical applications of refactoring detection.

1 Introduction

Refactoring is a well-known technique to improve the design of a system and enable its evolution [1]. In fact, existing studies [2], [3], [4], [5], [6] show that refactoring is frequently applied by development teams, and it is an important aspect of their software maintenance workflow.

Therefore, detecting refactoring activity in software projects is a valuable information to help researchers understand software evolution. For example, past studies used such information to shed light on important aspects of refactoring practice, such as the usage of refactoring tools [2], [7], the motivations driving refactoring [4], [5], [6], the risks of refactoring [4], [5], [8], [9], [10], and the impact of refactoring on code quality metrics [4], [5]. Moreover, it is often important to keep track of refactorings when performing source code evolution analysis because files, classes, or functions may have their histories split by refactorings such as *Move* or *Rename* [11].

Additionally, knowing which refactoring operations were applied in the version history of a system may help in several practical tasks. For example, in a study by Kim et al. [4], many developers mentioned the difficulties they face when reviewing or integrating code changes after large refactoring operations, which impact several code elements. Thus, developers might feel discouraged to refactor their code. If a tool is able to identify such refactoring operations, it can possibly resolve merge conflicts automatically. Moreover, diff visualization tools can also benefit from such information, presenting refactored code elements side-by-side with their corresponding version before the change. Another application for such information is adapting client code to a refactored version of an API it uses [12], [13]. If we are able to detect the refactorings that were applied to an API, we might be able to replay them on the client code automatically.

Given the importance of studying refactoring activity, we proposed RefDiff in previous work [14]. RefDiff is an automated approach that identifies refactoring operations performed in the version history of Java systems. By that time, our main goal was to provide a reliable tool to mine refactoring activity in a fully automated fashion, with better precision and recall than existing approaches. Since then, other approaches have emerged, such as RMiner [15], which enhanced precision to even higher standards. Today, the availability of such tools enables large-scale and in-depth empirical studies on refactoring practice [6], [11].

Nevertheless, despite the advancements in the field of refactoring detection, existing tools are centered in the Java language. Thus, we are still not able to mine refactoring activity in a vast amount of software repositories written in other programming languages. By restricting refactoring research to a single language, we may get a biased understanding of the reality. Interestingly, in the most recent edition of his refactoring book, Fowler changed all examples to use JavaScript [16], which corroborates the idea that refactoring practice in other languages should be discussed on equal footing with Java. Moreover, the practical applications of refactoring detection tools are hindered by the lack of

- *D. Silva and M.T. Valente are with the Department of Computer Science, Federal University of Minas Gerais, Av. Antônio Carlos, Belo Horizonte, MG 31270-010, Brazil.*
  E-mail: {danilofs, mtov}@dcc.ufmg.br.
- *J.P. da Silva is a developer and team lead at Quimbik, Inc.*
  E-mail: joao@jpribeiro.com.br.
- *G. Santos is with Federal University of Technology (UTFPR), Dois Vizinhos, PR 85660-000, Brazil.*
  E-mail: gustavosantos@utfpr.edu.br.
- *R. Terra is with the Department of Computer Science, Federal University of Lavras (UFLA), Lavras, MG 37200-000, Brazil.*
  E-mail: terra@dcc.ufla.br

support of other popular programming languages.

For all these reasons, in this paper we propose a multi-language refactoring detection approach, named as RefDiff 2.0, which is a redesign of its first version that introduces an extensible architecture. In RefDiff 2.0, the refactoring detection heuristics are fully implemented in a common core, and support for programming languages is provided by plug-in modules. As a way to validate this architecture, we implemented and evaluated extension modules for three mainstream programming languages with distinct characteristics: Java, JavaScript (a widely popular dynamic programming language, used mostly to build web applications) and C (a procedural programming language, used mostly to implement system software).

Additionally, we reworked the refactoring detection heuristics of RefDiff to significantly improve its precision when compared to our previous work. Now, RefDiff achieves 96.4% of precision and 80.4% of recall when evaluated in the Java dataset proposed by Tsantalis et al. [15], against 79.3% of precision and 80.2% of recall in its prior version. Moreover, RefDiff's precision is on par with RMiner, the current state-of-the-art in Java refactoring detection (96.4% vs. 98.8%). This is a key achievement because our approach is not specialized in a single language.

In summary, we deliver the following contributions in this work:

- A major extension of our refactoring detection approach proposed in previous work [14], which includes a redesign of its core to work with a language-independent model and improved detection heuristics.
- A publicly available implementation[1] of our approach, with out-of-the-box support for Java, C, and JavaScript.
- An evaluation of the precision and recall of RefDiff using a large scale dataset of refactorings performed in real-world Java open-source projects, comparing it with RMiner, a state-of-the-art tool for detecting refactorings in Java. As a byproduct of this evaluation, we also extend the dataset with new refactoring instances discovered by our tool.
- An evaluation of the precision and recall of RefDiff in real-world C and JavaScript open source projects.

The remainder of this paper is structured as follows. Section 2 describes related work, discussing existing refactoring detection approaches. Section 3 presents the proposed approach in details. Section 4 describes the design and results of a large scale evaluation of RefDiff in Java projects. Section 5 describes the design and results of an evaluation of RefDiff in C and JavaScript projects. Section 6 discusses challenges and limitations. Last, Section 7 presents final remarks and concludes the paper.

2 Background

Empirical studies on refactoring rely on means to identify refactoring activity. Thus, different techniques have been proposed and employed for this task. For example, Murphy-Hill et al. [2] collected refactoring usage data using a plug-in that monitors user actions in the Eclipse IDE, including calls to refactoring commands. Negara et al. [7] describe a tool, called CodingTracker, to infer refactorings from fine-grained code edits. They use this tool to study refactorings performed by 23 developers working in their IDEs during a total of 1,520 hours. The tool achieved a precision of 99.3% when evaluated with the automated Eclipse refactorings performed by the study participants. On a sample of both manual and automated refactorings, CodingTracker achieved a precision of 93% and a recall of 100%. However, CodingTracker requires the installation of a refactoring inference plugin in IDEs.

Other studies use metadata from version control systems to identify refactoring changes. For example, Ratzinger et al. [17] search for a predefined set of terms in commit messages to classify them as refactoring changes. In specific scenarios, a branch may be created exclusively to refactor the code, as reported by Kim et al. [5]. Another strategy is employed by Soares et al. [18]. They propose an approach that identifies behavior-preserving changes by automatically generating and running test-cases. While their approach is intended to guarantee the correct behavior of a system after refactoring, it may also be employed to classify commits as behavior-preserving. Moreover, many existing approaches are based on static analysis. This is the case of the approach proposed by Demeyer et al. [19], which finds refactored elements by observing changes in code metrics.

Static analysis is also frequently used to find differences in the source code by comparing two revisions [3], [14], [15], [20], [21], [22], [23]. Approaches based on comparing source code differences have the advantage of beeing able to identify refactoring operations applied in version histories. As RefDiff is one of these approaches, it can be directly compared with others within this category. In the next sections, we discuss RefDiff 1.0 and three other approaches.

2.1 RefDiff 1.0

The original version of RefDiff [14], which we will denote as RefDiff 1.0 throughout this paper, employs a combination of heuristics based on static analysis and code similarity to detect 13 well-known refactoring types. One of its distinguishing characteristic is the use of the classical TF-IDF similarity measure from information retrieval to compute code similarity. In our previous work, we evaluated RefDiff 1.0 using an oracle of 448 refactoring operations, distributed across seven Java projects. We built this oracle by deliberately applying refactorings in software repositories in a controlled manner. Although this strategy poses the risk of creating an artificial dataset, this way we assured this oracle was complete and could be used to compute both precision and recall. We compared our tool with three existing approaches, namely Refactoring Miner [3], Refactoring Crawler [20], and Ref-Finder [23]. Our approach achieved precision of 100% and recall of 88%, surpassing the three tools subjected to the comparison.

2.2 Refactoring Miner/RMiner 1.0

Refactoring Miner is an approach originally introduced by Tsantalis et al. [3], capable of identifying 14 high-

1. RefDiff and our evaluation data are public available at: https://github.com/aserg-ufmg/RefDiff

level refactoring types: *Rename Package/Class/Method*, *Move Class/Method/Field*, *Pull Up Method/Field*, *Push Down Method/-Field*, *Extract Method*, *Inline Method*, and *Extract Superclass/Interface*. In its original version, Refactoring Miner employs a lightweight algorithm, similar to the UMLDiff proposed by Xing and Stroulia [24], for differencing object-oriented models, inferring the set of classes, methods, and fields added, deleted or moved between two code revisions. Refactoring Miner was employed and evaluated in empirical studies on refactoring along its evolution. In the first study, using the version histories of JUnit, HTTPCore, and HTTPClient, Tsantalis et al. [3] reported 8 false positives for the *Extract Method* refactoring (96.4% precision) and 4 false positives for the *Rename Class* refactoring (97.6% precision). No false positives were reported for the remaining refactorings. In a second study that mined refactorings in 285 GitHub hosted Java repositories [6], we found found 1,030 false positives out of 2,441 refactorings (63% precision). However, we also evaluated Refactoring Miner using as a benchmark the dataset reported by Chaparro et al. [25], in which it achieved 93% precision and 98% recall.

In a recent study, Tsantalis et al. [15] proposed a major evolution of its tool, now named as RMiner (version 1.0). RMiner relies on an AST-based statement matching algorithm and a set of detection rules that cover 15 representative refactoring types. Its statement matching algorithm employes two techniques to be resilient to code restructuring during refactoring: abstraction, which deals with changes in statements' AST type due to refactorings, and argumentization, which deals with changes in sub-expressions within statements due to parameterization. To evaluate RMiner, the authors created a dataset with 3,188 real refactorings instances from 185 open-source projects. Using this oracle, the authors found that RMiner has a precision of 98% and recall of 87%, which was the best result so far, surpassing RefDiff 1.0, the previous state-of-the-art, which achieved precision of 75.7% and recall of 85.8% in this dataset.

2.3 Refactoring Crawler

Refactoring Crawler, proposed by Dig et al. [20], is an approach capable of finding seven high-level refactoring types: *Rename Package/Class/Method*, *Pull Up Method*, *Push Down Method*, *Move Method*, and *Change Method Signature*. It uses a combination of syntactic analysis to detect refactoring candidates and a reference graph analysis to refine the results.

First, Refactoring Crawler analyzes the abstract syntax tree of a program and produces a tree, in which each node represents a source code entity (package, class, method, or field). Then, it employs a technique known as *shingles encoding* to find similar pairs of entities, which are candidates for refactorings. Shingles are representations for strings with the following property: if a string changes slightly, then its shingles also change slightly. In a second phase, Refactoring Crawler applies specific strategies for detecting each refactoring type, and computes a more costly metric that determines the similarity of references between code entities in two versions of the system. For example, two methods are similar if the sets of methods that call them are similar, and the sets of methods they call are also similar. The strategies to detect refactorings are repeated in a loop until no new refactorings are found. Therefore, the detection of a refactoring, such as a rename, may change the reference graph and enable the detection of new refactorings.

The authors evaluated Refactoring Crawler comparing pairs of releases of three open-source software components: Eclipse UI, Struts, and JHotDraw. Such components were chosen because they provided detailed release notes describing API changes. The authors relied on such information and on manual inspection to build an oracle containing 131 refactorings. The reported results are: Eclipse UI (90% precision and 86% recall), Struts (100% precision and 86% recall), and JHotDraw (100% precision and 100% recall).

2.4 Ref-Finder

Ref-Finder, proposed by Prete et al. [22], [23], is an approach based on logic programming capable of identifying 63 refactoring types from the Fowler's catalog [1]. The authors express each refactoring type by defining structural constraints, before and after applying a refactoring to a program, in terms of template logic rules.

First, Ref-Finder traverses the abstract syntax tree of a program and extracts facts about code elements, structural dependencies, and the content of code elements, to represent the program in terms of a database of logic facts. Then, it uses a logic programming engine to infer concrete refactoring instances, by creating a logic query based on the constraints defined for each refactoring type. The definition of refactoring types also consider ordering dependencies among them. This way, lower-level refactorings may be queried to identify higher-level, composite refactorings. The detection of some types of refactoring requires a special logic predicate that indicates that the similarity between two methods is above a threshold. For this purpose, the authors implemented a block-level clone detection technique, which removes any beginning and trailing parenthesis, escape characters, white spaces, and return keywords and computes word-level similarity between the two texts using the longest common sub-sequence algorithm.

The authors evaluated Ref-Finder in two case studies. In the first one, they used code examples from the Fowler's catalog to create instances of the 63 refactoring types. The authors reported 93.7% recall and 97.0% precision for this first study. In the second study, the authors used three open-source projects: Carol, jEdit, and Columba. In this case, Ref-Finder was executed in randomly selected pairs of versions. From the 774 refactoring instances found, the authors manually inspected a sample of 344 instances and found that 254 were correct (73.8% precision).

It is worth noting that Ref-Finder and Refactoring Crawler require a full build of the program under analysis. Therefore, their usage is not recommended when mining refactorings from version histories in the large. In this case, it might be a challenge to build each release, due to missing external dependencies, for example. For that reason, our evaluation (Section 4) focus on comparing RefDiff with RMiner.

3 Proposed Approach

Our approach consists of two phases: Source Code Analysis and Relationship Analysis. In the first phase, Source Code

Analysis, we take as input two revisions of a system, v_1 and v_2, and build two models that represent their source code. Both models have the form of a tree, in which each node corresponds to a code element (classes, functions, etc.). In the second phase, Relationship Analysis, we compute a set R, which contains triples of the form (n_1, n_2, t), where n_1 is a code element from revision v_1, n_2 is a code element from revision v_2 and t is a relationship type. Such relationships may denote a high-level refactoring operation (move, rename, extract, etc.) or an exact correspondence between the code elements. For example, consider the diff between two revisions of a system depicted in Figure 1. Among other changes, the class `Calculator`, declared in revision 1, is renamed to `FpCalculator` in revision 2. This corresponds to a relationship of the type *Rename* between them. In the next sections, we describe in details each phase of our approach.

3.1 Phase 1: Source Code Analysis

The goal of this phase is to compute a language-independent model that represents the source code of the system, which we denote from now on as *Code Structure Tree* (CST). The CST is a tree-like structure that resembles an *Abstract Syntax Tree* (AST). However, in this representation we are only interested in coarse-grained code elements (e.g., classes and functions) that encompass a code region and may be referred by an identifier in other parts of the system.

To construct the CST, we need to parse the source code, generate the AST for the target programming language, and extract the necessary information. Thus, the decision of which types of AST nodes become CST nodes depends on the programming language. For example, in Java we represent classes, enums, interfaces, and methods as CST nodes. In contrast, local variables are not represented. Nevertheless, it is important to note that the granularity of the CST nodes determines the granularity of the relationships we are able to find, e.g., we can only find relationships between methods if we represent methods in the CST. Table 1 lists the types of AST nodes that are represented in the CST for each programming language supported by the current implementation of our approach.

TABLE 1
AST nodes that are represented in CSTs

Language	Node types
Java	class, enum, interface, and method
C	file and function
JavaScript	file, class, and function

Figure 2 exemplifies the transformation of the example system from Figure 1 into a corresponding CST. In revision 1, the class `Main` is declared with a single method `main` and the class `Calculator` contains two methods: `sum` and `min`. Note that these classes and methods become nodes in the CST for revision 1, preserving the same nesting structure of the source code. Analogously, the figure also depicts the CST for revision 2, which contains seven nodes in total (two classes and five methods).

Besides the representation of the code elements, the CST also embeds a simplified call graph and a type hierarchy graph of the nodes within the CST, that is, there are edges to represent whether a certain node n_1 calls n_2, or whether n_1 is a subtype of n_2. The first information is necessary to find *Extract* and *Inline* relationships between code elements, while the second is used to find inheritance-related relationships, such as *Pull Up* and *Push Down*.

Moreover, along with each node of the CST, we store the following information:

Identifier
: An identifier of the code element in its declared scope. The identifier is usually the name of the code element, but it may also contain additional information to avoid ambiguities. For example, the identifier of the class `Calculator` from Figure 2 is simply its name, but the identifier of the method `sum` is `sum(double,double)`, because there could be an overloaded method with a different signature.

Namespace
: An optional prefix that, along with the identifier, globally identifies the code element. This information only applies to top-level nodes and corresponds to the package or folder that the element is contained. For example, the namespace of the class `Calculator` from Figure 2 is `my.calc.`.

Node type
: A string that identifies the node type in the target language (class, function, method, etc.).

Parameters list
: An optional list of the name of the parameters, in the case the node corresponds to a method or function.

Tokenized source code
: The source code of the element in the form of a list of tokens. Here, we include all tokens in the code region that encompasses the complete declaration of the code element, including its name/signature. This information is necessary to compute the similarity between code elements, as explained in Section 3.3.

Tokenized source code of the body
: The source code of the body of the code element in the form of a list of tokens. Here we include only the tokens within the body of the code element, but not its name/signature. This information is also necessary to compute the similarity between code elements in the special cases of *Extract* and *Inline* relationships, as explained in Section 3.3.2. It is worth noting that this information is optional, as not every node has a body (e.g., abstract methods).

It is worth noting that we generate the CST only for source files that have been added, removed, or modified between revisions. Such information can be efficiently obtained from version control systems, without the need to analyze the content of all files within the repository. This way, we avoid a costly operation that might compromise

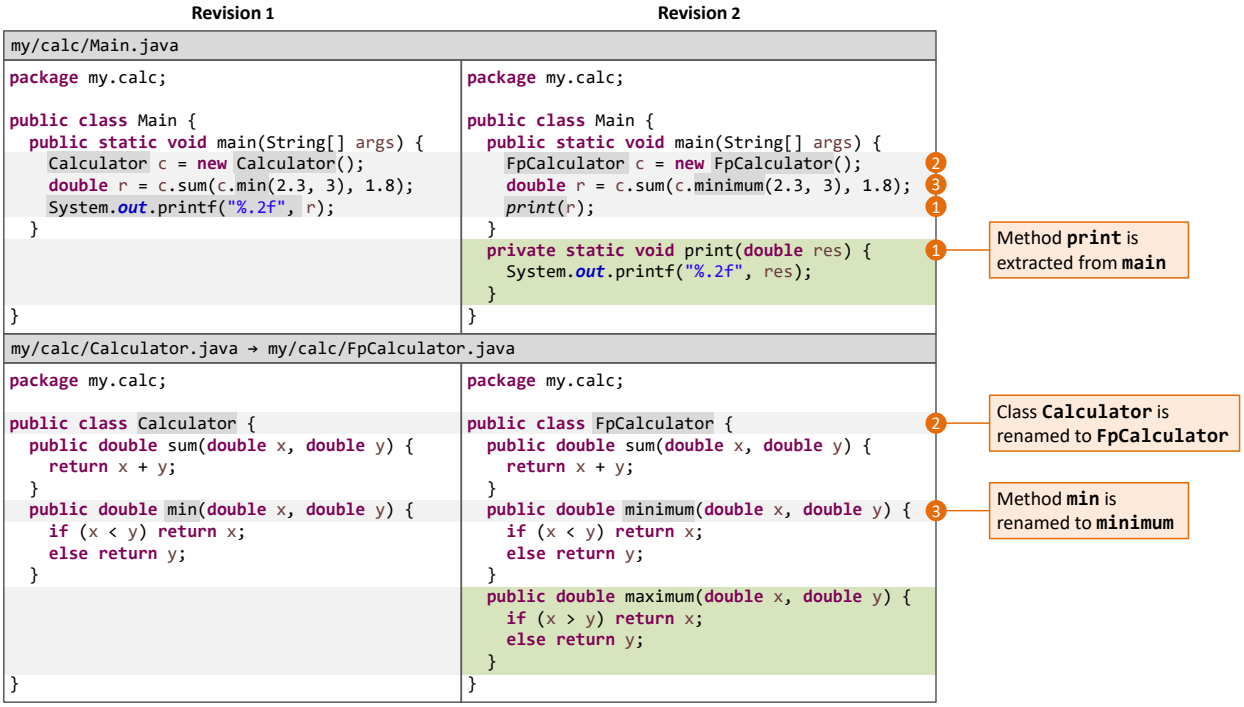


Fig. 1. Illustrative diff between two revisions of a system annotated with refactoring operations

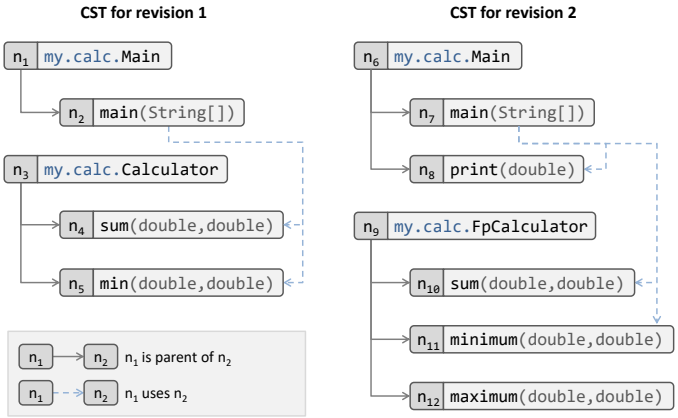


Fig. 2. CST of both revisions of the example system from Figure 1

the scalability of our approach, as large repositories contain thousands of source files, but only a small fraction of them change between revisions.

Although the construction of the CST is a language-specific process, from this point on, the approach is language-independent and relies only on information encoded in CSTs. This way, one is able to extend our approach to work with different programming languages only by implementing the *Source Code Analysis* module. To demonstrate this capability, we provide implementations for three programming languages: Java, C, and JavaScript.

3.2 Phase 2: Relationship Analysis

This phase takes as input the CST's of revisions v_1 and v_2 and outputs the set of relationships R. Let N_1 and N_2 be the sets of code elements from the CST's of v_1 and v_2 respectively. Each relationship $r \in R$ is a triple (n_1, n_2, t), where $n_1 \in N_1$, $n_2 \in N_2$, and t is a relationship type. The types of relationships are listed in the first column of Table 3, and can be subdivided into two groups:

- **Matching relationships**, which indicate that the node n_1 corresponds to n_2 in the subsequent revision. The possible matching relationships are *Same*, *Convert Type*, *Pull Up*, *Push Down*, *Change Signature*, *Move*, *Rename*, and *Move and Rename*. We say that a node n_1 matches with n_2 if exists a relationship $(n_1, n_2, t) \in R$ such that t is a matching relationship.
- **Non-matching relationships**, which indicate that either node n_1 is decomposed to create n_2, or node n_1 is incorporated into n_2. There are four non-matching relationships: *Extract Supertype*, *Extract*, *Extract and Move*, and *Inline*.

3.2.1 General algorithm to find relationships

Our approach employs the algorithm described in Figure 3 to find the relationships (i.e., to compute the set R). The procedure FINDRELATIONSHIPS has two parameters, t_1 and t_2, which are the root nodes of the CST's of both revisions. Initially, we define $R \leftarrow \emptyset$ as the set of relationships found so far (line 2). Additionally, we also define $M \leftarrow \emptyset$ as the set of pairs of matching nodes found so far (line 3). Then, we execute four subroutines:

1) In FINDMATCHINGSBYID, we recursively look for matching nodes that have the same identifier and parent, i.e., we assume that code elements with the same identifier and parent are the same. In the case of top-level nodes, which do not have parents, their namespace should be the same. Such assumption allows us to match many code elements at this step, reducing the number of possibilities that need to be

```
1: procedure FINDRELATIONSHIPS(t1, t2)
2:     R ← ∅
3:     M ← ∅
4:     FINDMATCHINGSBYID(t1, t2)
5:     FINDMATCHINGSBYSIM
6:     FINDMATCHINGSBYCHILDR
7:     RESOLVEMATCHINGS
8:     FINDNONMATCHINGREL
9:     return R
10:
11:    procedure FINDMATCHINGSBYID(p1, p2)
12:        for each (n1, n2) ∈ childr(p1) × childr(p2) do
13:            if id(n1) = id(n2) ∧ ns(n1) = ns(n2) then
14:                ADDMATCH(n1, n2)
15:            end if
16:        end for
17:    end procedure
18:
19:    procedure FINDMATCHINGSBYSIM
20:        for each (n1, n2) ∈ sortBySim(N⁻ × N⁺) do
21:            if findMatchRel(n1, n2) ≠ ∅ then
22:                ADDMATCH(n1, n2)
23:            end if
24:        end for
25:    end procedure
26:
27:    procedure FINDMATCHINGSBYCHILDR
28:        for each (n1, n2) ∈ sortBySim(N⁻ × N⁺) do
29:            if matchingChildr(n1, n2) > 1 ∧
                   nameSim(n1, n2) > 0.5 then
30:                ADDMATCH(n1, n2)
31:            end if
32:        end for
33:    end procedure
34:
35:    procedure RESOLVEMATCHINGS
36:        for each (n1, n2) ∈ M do
37:            R ← R ∪ findMatchRel(n1, n2)
38:        end for
39:    end procedure
40:
41:    procedure FINDNONMATCHINGREL
42:        for each (n1, n2) ∈ M1 × N⁺ do
43:            R ← R ∪ findExtractSupertype(n1, n2)
44:            R ← R ∪ findExtract(n1, n2)
45:            R ← R ∪ findExtractMove(n1, n2)
46:        end for
47:        for each (n1, n2) ∈ N⁻ × M2 do
48:            R ← R ∪ findInline(n1, n2)
49:        end for
50:    end procedure
51:
52:    procedure ADDMATCH(n1, n2)
53:        if n1 ∈ N⁻ ∧ n2 ∈ N⁺ then
54:            M ← M ∪ {(n1, n2)}
55:            FINDMATCHINGSBYID(n1, n2)
56:        end if
57:    end procedure
58:
59: end procedure
```

Fig. 3. Algorithm to find relationships

checked in the next steps. The procedure consists of a loop that pairs the children of the nodes received as arguments and calls the procedure ADDMATCH whenever a matching is found (line 13). On its turn, ADDMATCH (lines 46-51) adds a pair of matching nodes to M and calls FINDMATCHINGSBYID again to look for matchings on their children, completing the recursion. The matching pairs found in this step will be resolved to *Same* and *Convert type* relationships later (see step 4).

2) In FINDMATCHINGSBYSIM, we look for matching nodes based on code similarity. The goal is to find *Change Signature*, *Pull Up*, *Push Down*, *Move*, *Rename*, and *Move and Rename* relationships. The procedure iterates over the unmatched pairs of nodes sorted by similarity in descending order. We use the notation N^- to denote the set of unmatched nodes from t_1 (presumably deleted) and N^+ to denote the set of unmatched nodes from t_2 (presumably added). For each pair (n_1, n_2), the procedure checks if it meets the conditions (specified in the second column of Table 3) for any matching relationship by calling $findMatchRel(n_1, n_2)$. This function returns a singleton containing a matching relationship or an empty set if none of the conditions are met. Last, the ADDMATCH subroutine is called in the case of a matching (line 24). The conditions to find those relationships and the $\mathrm{sortBySim}$ function rely on a code similarity metric, which is described in details in Section 3.3.
3) In FINDMATCHINGSBYCHILDR, we look for matching nodes based on matchings of their children and name similarity. Once again, the procedure iterates over the unmatched pairs of nodes sorted by similarity in descending order. For each pair (n_1, n_2), if n_1 has more than one children that match with n_2's children and their names are similar, then we consider it a match. The $\mathrm{nameSim}$ function, used to compute the similarity between names, is described in details in Section 3.3.1. This heuristic is intended to cover the cases when a code element (e.g., a class) is moved (and/or renamed) and it is also subjected to many additions or removals of its members, so that its similarity with its matching pair is not enough to yield a match in the previous step. Failing to detect that a class has been moved (or renamed) may yield several incorrect *Move* relationships between its members before and after the change.
4) In RESOLVEMATCHINGS, we add the relationships corresponding to the matching pairs found at steps 1, 2, and 3 to R. The procedure iterates over the elements of M and calls *findMatchRel* to find which relationship type holds between n_1 and n_2 (according to the conditions defined in Table 3). By the end of this step, R contains all matching relationships found. The rationale for postponing the resolution of the relationship type is discussed in Section 3.2.2.
5) In FINDNONMATCHINGREL, we look for non-matching relationships. First, we iterate over the pairs of matched/unmatched nodes, i.e., $M_1 \times N^+$,

to look for *Extract Supertype*, *Extract* and *Extract and Move* relationships. Similarly, we also iterate over the pairs of unmatched/matched nodes ($N^- \times M_2$) to search for *Inline* relationships. The functions $findExtractSupertype$, $findExtract$, $findExtractMove$, and $findInline$ check the preconditions for the corresponding relationship types, according to Table 3. After this last step, R contains all matching and non-matching relationships between CST nodes of both revisions.

Figure 4 shows the relationships we find after running RefDiff in the example from Figure 1. Each relationship is represented by an edge connecting nodes from the left and right CSTs. There are three relationships of the type *Same*, involving the code elements whose identifiers do not change: the class `Main` and the methods `main` and `sum`. Two of the relationships are of type *Rename*, indicating that the class `Calculator` is renamed to `FpCalculator`, and the method `min` is renamed to `minimum`. Moreover, there is an *Extract* relationship indicating that the method `print` is extracted from `main`. Finally, we can also note that two nodes, n_8 and n_{12}, are not involved in matching relationships. Thus, we classify them as added code elements. In this example, as every node on the left side is matched, there are no deleted code elements.

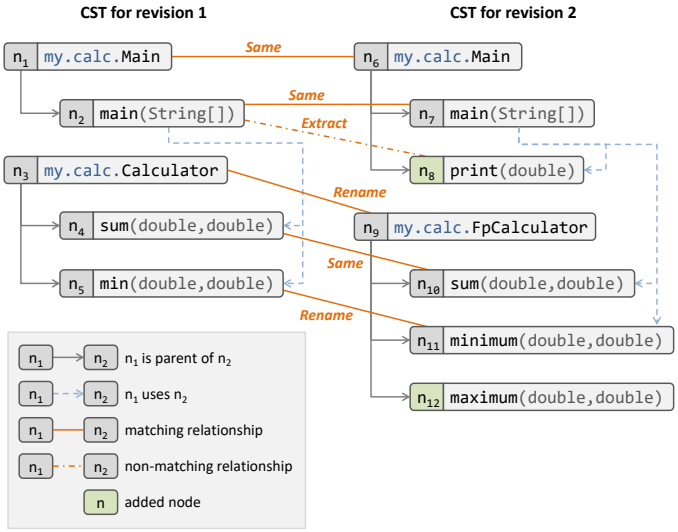


Fig. 4. Relationships found in the example from Figure 1

3.2.2 Dependent and conflicting relationships

In some cases, correctly finding a relationship depends on finding a prior relationship. For example, consider the relationship $(n_5, n_{11}, Rename)$ in Figure 4 (method `min` renamed to `minimum`). The conditions for this relationship includes the clause $\pi(n_5)' = \pi(n_{11})$, which means that the matching node of the parent of n_5 should be equal to the parent of n_{11} (see Table 3, *Rename* row). This clause only yields true after the matching pair (n_3, n_9) is added to M, i.e., after we find out that `Calculator` is renamed to `FpCalculator`. In fact, if we call $findMatchRel(n_5, n_{11})$ before M contains (n_3, n_9), we would incorrectly classify it as a *Move and Rename* relationship. To address this issue, we only resolve the actual relationship types in steps 4 and 5, after all matching pairs are found (note that in steps 1, 2, and 3 we record the matching pairs in M, purposely ignoring the type of relationship).

Another issue which we may face when looking for relationships are conflicts, i.e., two or more matching relationships hold for the same code element (according to conditions from Table 3). For instance, in the example from Figure 4, the conditions for *Rename* yield true for the pair of methods `min` and `minimum` because their source code are similar and their parents match. However, this is also the case for the pair of methods `min` and `maximum`, whose bodies are also similar. We cannot match the same node twice, thus, we must decide upon which relationship we will accept and discard the other one. This issue is addressed in procedures FINDMATCHINGSBYSIM and FINDMATCHINGSBYCHILDR by using the sortBySim function to sort the potential matching pairs, enforcing that we take first the most likely matches. The sortBySim function relies on a similarity metric, which we discuss in details in Section 3.3. After a matching pair (n_1, n_2) is added to M, no more matchings involving n_1 or n_2 are accepted, because ADDMATCH procedure checks that $n_1 \in N^- \wedge n_2 \in N^+$ (line 47).

3.3 Code Similarity

A key element of our approach to find relationships, as previously mentioned, is computing the similarity between code elements (i.e., CST nodes). The first step to compute this similarity is to represent their source code as a multiset (or bag) of tokens. A multiset is a generalization of the concept of a set, but it allows multiple instances of the same element. The multiplicity of an element is the number of occurrences of that element within the multiset. This representation provides two advantages for our approach. First, it makes the CST simpler and less coupled to the syntax of programming languages, because we do not need to represent each statement (or AST node) from the source code. Second, we can apply set operations to the bag of tokens, such as subtraction, which is important to detect *Extract* relationships, as we will discuss in Section 3.3.2.

Formally, a multiset can be defined in terms of a multiplicity function $m : U \to \mathbb{N}$, where U is the set of all possible elements. In other words, $m(t)$ is the multiplicity of the element t in the multiset. Note that the multiplicity of an element that is not in the multiset is zero. For example, Figure 5 depicts the transformation of the source code of three methods (`sum`, `min`, and `power`), of the class `Calculator`, into multisets of tokens. In this figure, the multiplicity function m for each method is represented in a tabular form. For example, the multiplicity of the token `y` in method `min` is two (i.e., $m_{\texttt{min}}(\texttt{y}) = 2$), whilst the multiplicity of the token `if` in method `power` is zero (i.e., $m_{\texttt{power}}(\texttt{if}) = 0$).

After extracting a multiset of tokens, we also compute a weight for each token of the source code. In fact, some tokens are more important than others to discriminate a code element. For example, in Figure 5, all three methods contain the token `return`. In contrast, only one method (`power`) contains the token `Math`. Therefore, the later is

TABLE 2
Definitions used in the Algorithm from Figure 3 and in the conditions from Table 3

M_1	the set of nodes from N_1 that matches with a node from N_2	$\text{name}(n)$	simple name of the code element n
M_2	the set of nodes from N_2 that matches with a node from N_1	$\text{id}(n)$	identifier of the code element n
N^-	the set of unmatched nodes from N_1 ($N_1 \setminus M_1$)	$\text{nType}(n)$	node type of the code element n
N^+	the set of unmatched nodes from N_2 ($N_2 \setminus M_2$)	$\text{subtype}(n_1, n_2)$	n_1 is subtype of n_2
n'	the code element that matches with n in the other revision	$\text{uses}(n_1, n_2)$	n_1 uses n_2
$\pi(n)$	parent of a node n (it may be a namespace or a CST node)	$\text{sim}(n_1, n_2)$	code similarity between n_1 and n_2
$\text{ns}(n)$	namespace of the code element n	$\text{nameSim}(n_1, n_2)$	name similarity between n_1 and n_2
$\text{childr}(n)$	set of children of n in the CST	$\text{sim}_\text{x}(n_1, n_2)$	extract similarity between n_1 and n_2
		$\text{sortBySim}(S)$	elements of S sorted by sim descending

TABLE 3
Relationship types and the conditions to find them

Relationship type	Conditions
	$(n_1, n_2) \in N^- \times N^+$, such that:
Same	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \pi(n_1)' = \pi(n_2)$
Convert Type	$\text{nType}(n_1) \neq \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \pi(n_1)' = \pi(n_2)$
Pull Up	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \text{subtype}(\pi(n_1)', \pi(n_2))$
Push Down	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) = \text{id}(n_2) \wedge \text{subtype}(\pi(n_2), \pi(n_1)')$
Change Signature	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{id}(n_1) \neq \text{id}(n_2) \wedge \text{name}(n_1) = \text{name}(n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Move	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) = \text{name}(n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Rename	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) \neq \text{name}(n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
Move and Rename	$\text{nType}(n_1) = \text{nType}(n_2) \wedge \text{name}(n_1) \neq \text{name}(n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}(n_1, n_2) > 0.5$
	$(n_1, n_2) \in M_1 \times N^+$, such that:
Extract Supertype	$\exists (n_3, n_4, PullUp) \in R\, (n_1 = \pi(n_3) \wedge n_2 = \pi(n_4))$
Extract	$\text{uses}(n_1', n_2) \wedge \pi(n_1)' = \pi(n_2) \wedge \text{sim}_\text{x}(n_2, n_1) > 0.5$
Extract and Move	$\text{uses}(n_1', n_2) \wedge \pi(n_1)' \neq \pi(n_2) \wedge \text{sim}_\text{x}(n_2, n_1) > 0.5$
	$(n_1, n_2) \in N^- \times M_2$, such that:
Inline	$\text{uses}(n_1, n_2') \wedge \text{sim}_\text{x}(n_1, n_2) > 0.5$

a better indicator of similarity between methods than the former.

In order to take this into account, we employ a variation of the TF-IDF weighting scheme [26], which is a well-known technique from information retrieval. TF-IDF, which is the short form of *Term Frequency–Inverse Document Frequency*, reflects how important a term is to a document within a collection of documents. In the context of code elements, we consider a token as a term, and a code element as a document. Let E be the set of all code elements and n_t be the number of elements in E that contains the token t. The Inverse Document Frequency (idf), is defined as:

$$idf(t) = \log(1 + \frac{|E|}{n_t}) \tag{1}$$

Note that the value of $idf(t)$ decreases as n_t increases, because the more frequent a token is among the collection of code elements, the less important it is to distinguish them. For example, in Figure 5, the token `y` occurs in two methods (`sum` and `min`). Thus, its idf is:

$$idf(\text{y}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{2}) = 0.398$$

On the other hand, the token `else` occurs in one method (`min`), and therefore its idf is:

$$idf(\texttt{else}) = \log(1 + \frac{|E|}{n_t}) = \log(1 + \frac{3}{1}) = 0.602$$

Last, to compute the similarity between two code elements e_1 and e_2, we use a generalization of the Jaccard coefficient, known as weighted Jaccard coefficient [27]. Let U be the set of all possible tokens and $m_i : U \to \mathbb{N}$ be the multiplicity function representing the multiset of tokens of a code element e_i. We define the similarity between e_1 and e_2 by the following formula:

$$\text{sim}(e_1, e_2) = \frac{\sum_{t \in U} \min(m_1(t), m_2(t)) \times idf(t)}{\sum_{t \in U} \max(m_1(t), m_2(t)) \times idf(t)} \tag{2}$$

The rationale behind this formula is that the similarity is at maximum (1.0) when the multiset of tokens representing e_1 and e_2 contain the same tokens with the same cardinality. In contrast, if the multisets contain no tokens in common, the similarity is zero. Additionaly, tokens with higher idf will have a higher weight.

Source code of a class

```
public class Calculator {

  public int sum(int x, int y) {
    return x + y;
  }

  public int min(int x, int y) {
    if (x < y) return x;
    else return y;
  }

  public double power(int b, int e) {
    return Math.pow(b, e);
  }
}
```

⇒

Multiset of tokens for each method

Token t	$m_{\texttt{sum}}(t)$	$m_{\texttt{min}}(t)$	$m_{\texttt{power}}(t)$	n_t
`return`	1	2	1	3
`x`	1	2	0	2
`+`	1	0	0	1
`y`	1	2	0	2
`;`	1	2	1	3
`if`	0	1	0	1
`(`	0	1	1	2
`<`	0	1	0	1
`)`	0	1	1	2
`else`	0	1	0	1
`Math`	0	0	1	1
`.`	0	0	1	1
`pow`	0	0	1	1
`b`	0	0	1	1
`,`	0	0	1	1
`e`	0	0	1	1

Fig. 5. Transformation of the body of methods into a multiset of tokens

3.3.1 Name similarity

Besides relying on the code similarity, our algorithm also depends on the function $\mathrm{nameSim}(n_1, n_2)$ in Step 3. This function denotes the similarity between the names of the code elements n_1 and n_2. For computing $\mathrm{nameSim}$, we first decompose the identifiers of n_1 and n_2 into their composing words. Specifically, we split camel case (e.g., `myIdentifier`) or snake case patterns (e.g., `my_identifier`) . For example, `SomeLong_Name` yields three terms: `Some`, `Long`, and `Name`. Then, we compute $\mathrm{nameSim}$ using the same formula from sim (see Equation 2), but in this case, each multiset of tokens contains the terms composing the identifiers of n_1 and n_2.

3.3.2 Extract similarity

While the similarity function sim presented previously is suitable to compute whether two code elements are similar, it is not appropriate to assess whether a code element is extracted from another one, because their source code may be significantly different on their entirety. However, if a method e_2 is extracted from e_1, we expect that part of the code of e_1 is moved to e_2. Therefore, the source code of the body of e_2 should be similar to the source code removed from e_1. Additionally, not all code removed from e_1 may have been moved to e_2, i.e., some parts of the code may have been extracted to other locations or simply deleted. To be less susceptible to this issue, our similarity index for *Extract* relationships rely on the following assumption: the code from the body of e_2 should be mostly contained in the code removed from e_1.

Thus, to compute the extract similarity, first we need to compute the code removed from e_1. As we represent the source code as multisets of tokens, we are able to use the subtract operation to achieve this goal. Let m_1 be the multiset of tokens of e_1 before the change and m_1' be the multiset of tokens of e_1 after the change. The subtract operation between both multisets, which we denote by $m_1' \setminus m_1$, yields a new multiset m_1^- defined by the following multiplicity function:

$$m_1^-(t) = \max(0, m_1'(t) - m_1(t)) \tag{3}$$

Besides computing the code removed from e_1, we need to measure if it is contained within e_2. Thus, we employ a variation of the weighted Jaccard coefficient introduced previously (see Equation 2), which is defined as:

$$\mathrm{sim}_{\subseteq}(m_1, m_2) = \frac{\sum_{t \in U} \min(m_1(t), m_2(t)) \times idf(t)}{\sum_{t \in U} m_1(t) \times idf(t)} \tag{4}$$

where m_1 and m_2 are multisets (defined by their multiplicity functions). In this variation, we change the denominator of Equation 2 to include only the multiplicity of the tokens from the first multiset (not their union). This way, the similarity is at maximum (1.0) when m_1 is a subset of m_2, even if both multisets are not identical. In contrast, the similarity is zero when the intersection between m_1 and m_2 is empty.

Given these definitions, we are able to define the extract similarity index, $\mathrm{sim}_{\mathrm{x}}$, as:

$$\mathrm{sim}_{\mathrm{x}}(e_1, e_2) = \mathrm{sim}_{\subseteq}(m_2, m_1^-) \tag{5}$$

where m_1^- is the multiset representing the code removed from e_1 ($m_1 \setminus m_1'$) and m_2 is the multiset representing the source code of the body of e_2. The rationale behind this formula is that the similarity is at maximum when m_2 is a subset of m_1^-, addressing the previously described heuristic.

3.3.3 Inline similarity

The similarity index for computing *Inline* relationships is analogous to the *Extract* similarity index. If a code element e_1 is inlined into a code element e_2, we expect that the code from the body of e_1 should be mostly contained in the code added to e_2. Specifically, we define the inline similarity index, $\mathrm{sim}_{\mathrm{i}}$, as:

$$\mathrm{sim}_{\mathrm{i}}(e_1, e_2) = \mathrm{sim}_{\subseteq}(m_1, m_2^+) \tag{6}$$

where m_1 is the multiset representing the source code of the body of e_1 and m_2^+ is the multiset representing the code

added to e_2 ($m'_2 \setminus m_2$). Such similarity index is at maximum (1.0) when m_1 is a subset of the added code (m_2^+).

3.3.4 Ignoring parameters and return keywords

When retrieving the tokenized source code of the body of a code element, some tokens are ignored to avoid that syntactical constructs necessary to its declaration introduce noise when computing the *Extract* or *Inline* similarity index. For example, suppose we take the refactoring operation #1 depicted in Figure 1: `print` is extracted from `main`. The body of the new method `print` contains a single statement:

```
System.out.printf("%.2f", res);
```

All the tokens within this method are present in `main` before the extraction, except the identifier `res`, which is a declared parameter of `print`. In fact, in the original statement, a variable `r` is used in place of `res`. To be less susceptible to such differences, we omit all occurences of parameters in the tokenized source code of the body. Similarly, occurences of `return` keywords are also ignored because they may be introduced when turning the extracted code into a method. It is worth noting that discarding such tokens is of responsability of the source analysis module. Thus, specific rules may be implemented according to the peculiarities of the programming language.

3.4 Implementation details

RefDiff implementation consists of a core module and language plugins:

- **`refdiff-core`**: implements our core algorithm and contains common data types to represent CSTs and interfaces to implement source code analysis (i.e., generation of CSTs) for each programming language. Currently, this module contains 3,103 lines of code, implemented in Java.
- **`refdiff-java`**: language plugin for Java, which relies on the Eclipse JDT library for parsing and analyzing Java code.[2] This module contains 1,137 lines of code.
- **`refdiff-c`**: language plugin for C, using the Eclipse CDT library.[3] This module contains 615 lines of code.
- **`refdiff-js`**: language plugin for JavaScript, using the Babel parser[4] and with 689 lines of code.

To add support to a new programming language, one must implement the `LanguagePlugin` interface, which defines two methods: `parse`, which builds the CST given a set of source files, and `getAllowedFilesFilter`, which returns an object with a list of allowed file extensions and an optional list of ignored file name suffixes. For example, **`refdiff-js`** ignores file names that end with `.min.js`, which are usually generated code.

When compared to existing refactoring detection approaches, RefDiff's design has the advantage of being loosed coupled to the syntax of Java (and of any other programming language). For example, RMiner, which is a Java-based approach, relies on a statement matching algorithm and applies two techniques to enable matching of statements that are not textually identical: Abstraction and Syntax-aware replacements of AST nodes [15]. Both techniques manipulate syntactic constructs of the Java language, such as return statements, variable declarations, assignments, method invocations, conditional statements, class instantiations, types, literals, operators, and others. Thus, when adapting these techniques to other programming language, tool builders should carefully consider its particular syntactic constructs. On the other hand, RefDiff's similarity comparison is based on tokenized code. Therefore, it does not depend on the AST nodes of any given language. As another example, Java code is structured with classes, which contains methods and attributes, and RMiner detection rules are tightly based on this structure. In contrast, JavaScript code contains functions inside functions with arbitrary levels of nesting. RefDiff is able to deal with both languages because CSTs do not assume any particular hierarchical structure between different types of nodes.

In summary, we do not claim that existing approaches cannot not be extended to other languages, but that would require a non-trivial effort. By making fewer assumptions about the syntax of the target language we facilitate multi-language support. Note that the implemented language plugins have small code bases (between 615–1137 lines of code).

4 Evaluation with Java Projects

In this section, we evaluate the precision and recall of our approach using a recently proposed dataset of refactorings performed in real-world Java open-source projects. We also compare its precision and recall with RMiner—the current state-of-the-art tool for detecting refactorings in Java—and RefDiff 1.0, the previous version of our tool. First, we present our evaluation design (Section 4.1) and then we present the results (Section 4.2).

4.1 Evaluation Design

To evaluate the precision and recall of RefDiff 2.0 in Java we initially use an oracle proposed by Tsantalis et al. [15]. This oracle includes 3,188 manually-validated refactoring instances, detected in 538 commits from 185 open-source projects, and covering 15 refactoring types. It is important to emphasize that most commits contain non-refactoring changes interleaved with refactorings, which is the most challenging scenario for refactoring detection tools. In our evaluation, we also compare RefDiff's precision and recall against RMiner (version 1.0). For the purpose of the comparison, we restricted the oracle to 11 refactoring types supported by both tools. Specifically, we excluded *Change Package*, *Move Field*, *Push Down Field* and *Pull Up Field* from the analysis as they are not supported by RefDiff. Moreover, *Convert Type* and *Change Signature*, although supported by RefDiff, are not evaluated because they are not covered by the oracle. In total, our modified oracle contains 3,031 confirmed refactoring instances. Additionally, it also contains 704 refactoring instances classified as false positives in the process of manual validation performed by Tsantalis

2. https://www.eclipse.org/jdt/
3. https://www.eclipse.org/cdt/
4. https://babeljs.io/docs/en/babel-parser

```
213     -   protected void prepareOnAffectedNodes(TxInvocationContext<?> ctx, PrepareCommand command, Collection<Address> recipients, bool
        211 +   protected void prepareOnAffectedNodes(TxInvocationContext<?> ctx, PrepareCommand command, Collection<Address> recipients) {
214     212         try {
215     213            // this method will return immediately if we're the only member (because exclude_self=true)
216     -          RpcOptions rpcOptions;
217     -          if (sync) {
218     -             rpcOptions = rpcManager.getRpcOptionsBuilder(ResponseMode.SYNCHRONOUS_IGNORE_LEAVERS, DeliverOrder.NONE).build();
219     -          } else {
220     -             rpcOptions = rpcManager.getDefaultRpcOptions(false);
221     -          }
222     -          Map<Address, Response> responseMap = rpcManager.invokeRemotely(recipients, command, rpcOptions);
223     -          checkTxCommandResponses(responseMap, command);
        214 +          Map<Address, Response> responseMap = rpcManager.invokeRemotely(recipients, command, createPrepareRpcOptions());
        215 +          checkTxCommandResponses(responseMap, command, (LocalTxInvocationContext) ctx, recipients);
224     216         } finally {
225     217            transactionRemotelyPrepared(ctx);
226     218         }
227     219      }

        417 +   protected RpcOptions createPrepareRpcOptions() {
        418 +      return cacheConfiguration.clustering().cacheMode().isSynchronous() ?
        419 +            rpcManager.getRpcOptionsBuilder(ResponseMode.SYNCHRONOUS_IGNORE_LEAVERS, DeliverOrder.NONE).build() :
        420 +            rpcManager.getDefaultRpcOptions(false);
        421 +   }
```

Fig. 6. Illustrative diff of an *Extract Method* refactoring considered as true positive by the validators, taken from commit ce4f629 from *infinispan* project.

TABLE 4
Java precision and recall results

		RefDiff 1.0		RefDiff 2.0		RMiner 1.0	
Refactoring Type	#	Precision	Recall	Precision	Recall	Precision	Recall
Move Class	1100	0.999	0.881	0.999	0.970	1.000	0.925
Move Method	319	0.322	0.746	0.871	0.803	0.955	0.658
Move and Rename/Rename Class	95	0.897	0.642	0.922	0.874	0.983	0.621
Rename Method	350	0.855	0.811	0.946	0.694	0.978	0.771
Extract Interface	24	0.769	0.417	0.875	0.875	1.000	0.833
Extract Superclass	70	1.000	0.157	1.000	0.743	0.958	0.971
Pull Up Method	91	0.806	0.593	0.974	0.824	1.000	0.791
Push Down Method	40	0.950	0.475	0.950	0.950	1.000	0.825
Extract/Extract and Move Method	1037	0.904	0.833	0.962	0.663	0.985	0.768
Inline Method	122	0.842	0.787	0.957	0.721	0.990	0.795
Total	3248	0.792	0.802	0.964	0.804	0.988	0.813

et al. [15]. These instances are used to detect false positives reported by RefDiff, as described in the next paragraph.

First, we run RefDiff on each commit of the oracle. For each detected refactoring r we checked whether r is in the oracle, which may yield three outcomes: (i) if r is a confirmed refactoring from the oracle, then it is a true positive; (ii) if r is a false refactoring from the oracle, then it is a false positive; (iii) otherwise, r was inspected by two authors of this paper to assess whether it is a false positive or a true positive not covered by the oracle. This extra manual validation is needed because the initial oracle must not be granted as complete, i.e., including all refactorings performed in the set of analysed commits. Specifically, it was constructed using a triangulation approach, based on an initial list of refactorings produced by RMiner 1.0 and RefDiff 1.0. For this reason, it might miss true refactorings only detected by RefDiff 2.0.

After following this procedure, RefDiff 2.0 detected 263 new refactoring instances (i.e., not listed in the initial oracle), which were validated by two paper's authors, called here validators. In the case of 175 refactorings (66%), the validators agreed on their classification, including 138 refactorings labelled as true positives by both validators and 37 labelled as false positives. After this initial and independent validation, the validators discussed together the remaining 88 cases (34%), to reach an agreement. As a result, 79 refactorings were considered true positives and 9 refactorings were classified as false positives. Figure 6

shows an example of a refactoring identified by RefDiff that both validators classified as true positive. In this case, a developer extracted method `createPrepareRpcOptions` from method `prepareOnAffectedNodes`.

In total, after completing the manual validation, 217 new refactorings instances were classified as true positives and therefore included in the oracle. The expanded oracle includes 3,248 refactoring instances (7.19% more than the initial one) and it is publicly available at RefDiff's GitHub repository.[5]

4.2 Results and discussion

Table 4 shows the precision and recall results for RefDiff 2.0 and RMiner 1.0 using the oracle described in the previous section. The overall precision and recall of RefDiff 2.0 is 96.4% and 80.4%, respectively. Precision ranges from 87.1% (*Move Method*) to 100.0% (*Extract Superclass*), and it is above 90% for 8 out of 10 refactoring types. Recall ranges from 66.3% (*Extract Method*) to 97.0% (*Move Class*), and it is above 80% for 6 out of 10 refactoring types.

4.2.1 Comparison with RefDiff 1.0

We also show in Table 4 the results obtained with RefDiff 1.0 in this oracle. Note that overall precision is significantly improved (from 79.2% to 96.4%). Moreover, RefDiff 2.0 has less variation on recall across refactoring types. We can list five improvements over RefDiff 1.0 that justify such results.

- In RefDiff 2.0, we find moved/renamed types (e.g., classes) based on matched members (step 3 of our algorithm). This heuristic was introduced aiming to reduce the number of false negatives for class moves/renames, which also reduces the number of false positives for *Move Method*.
- We compute the removed and added code using set operations to improve *Extract* and *Inline* similarity functions. For example, our *Extract* similarity function compares the body of an extracted method with the code removed from the original method, strengthening the preconditions to detect *Extract* relationships. Similarly, our *Inline* similarity function compares the body of an inlined method with the code added to its destination.
- Ignoring parameters/arguments and return keyword (Section 3.3.4) is also an improvement over RefDiff 1.0, making *Extract* and *Inline* similarity less sensitive to code changes related to the mechanics of the refactoring.
- *Pull Up* and *Push Down* rules no longer include body similarity comparison. Additionally, *Extract Supertype* also drops similarity comparison and it was rewritten based on *Pull Up* rule. These changes improved both precision and recall of these refactoring types.
- While RefDiff 1.0 relied on a set of thresholds, which were calibrated for each refactoring type, in RefDiff 2.0 we use a single similarity threshold, defined as 0.5 by default. We acknowledged that relying on user-defined thresholds or thresholds calibration is not ideal, as advocated by Tsantalis et al. [15]. Thus, in RefDiff 2.0 we emphasized the aforementioned improvements to our algorithm, making it is less sensitive to similarity thresholds. In fact, we achieved better precision for all refactoring types, even without calibration. We only lost recall for *Rename Method*, *Extract Method* and *Inline Method*. We attribute this fact to the very low thresholds set for these refactoring types (between 0.1 and 0.3).

5. https://github.com/aserg-ufmg/RefDiff

4.2.2 Comparison with RMiner 1.0

Table 4 also shows the results of RMiner 1.0, which achieves 98.8% of overall precision (ranging from 95.5% to 100.0%) and 81.3% of overall recall (ranging from 64.1% to 97.1%). When we analyze individual refactoring types, RefDiff's precision is lower in all but one refactoring type (*Extract Superclass*). However, recall is higher in 6 refactoring types. In summary, both tools have very similar total recall, but RMiner's precision is slightly higher. We can list at least three differences between RefDiff and RMiner that might explain the differences in the results.

- Unlike RefDiff, we believe RMiner does not account for methods moved to added classes, nor methods moved from deleted classes, as RMiner's detection rule for *Move Method* includes the clauses $(td_a, td'_a) \in TD^{=}$ and $(td_b, td'_b) \in TD^{=}$ [15]. Many of the false negatives for *Move Method* involve such scenarios, which explains the lower recall for RMiner.
- Both approaches find moved/renamed types (e.g., classes) based on matched members (step 3 of our algorithm). However, ReffDiff's detection rule requires that a pair of candidate types (t_1, t_2) have more than one children in common, while RMiner's rule is more strict, requiring that either all members of t_1 are in t_2, or all members of t_2 are in t_1. Additionally, RefDiff also finds moved/renamed types by similarity. These might be the reasons for RMiner's lower recall for *Move and Rename/Rename Class*.
- They use very different approaches for computing code similarity. While RefDiff relies on tokenized code and a TF-IDF based similarity function, RMiner relies on a statement matching algorithm and syntax-aware replacement of AST nodes. Such difference potentially impacts precision and recall for several refactoring types, and might be an advantage factor for RMiner.

Despite the aforementioned differences, we emphasize that RefDiff and RMiner have much in common:

- Both approaches match elements by full name/signature in their first step.
- Many of the refactoring detection rules are similar.
- Both approaches enforce an order of detection between refactoring types and use a best match strategy to choose between conflicting refactoring candidates.
- RefDiff 2.0 included an heuristic to find moved/renamed types (e.g., classes) based on matched members, which is similar to RMiner's detection rule.
- Ignoring parameters/arguments and return keyword (Section 3.3.4) serves a similar purpose to

the argumentization and abstraction techniques proposed by RMiner.

4.3 Execution time

Besides comparing precision and recall, we also compared the execution time of both RefDiff 2.0 and RMiner 1.0. For this purpose, we ran both tools using the same computer (an Intel Core i5-750 with 8GB of RAM and a 7200 RPM HDD) and measured the time spent in the analysis of each of the 538 commits from our oracle.[6] Figure 7 shows a violin plot of the execution time per commit for both tools, using a log-10 scale. We can note that the median is lower for RMiner (109 ms vs. 157 ms), but RefDiff has less variation in the execution times. For example, the maximum execution time for RMiner was 85s, at commit ab98bca from *java-algorithms-implementation*, whilst RefDiff spent 10s at maximum, in commit 4baf0a4 from *aws-sdk-java*. Nevertheless, both tools analyze the majority of the commits in less than one second and are viable for practical use. It is worth noting that we executed both RefDiff and RMiner using their file-based API, which reads a list of files directly from disk. This means that the time to clone or checkout revisions from git repositories is not included in our measurements. However, we do not expect significant differences between both tools when using their git-based API, which includes services from mining refactorings directly from git repositories. The reason is that both tools retrieve only the necessary files using the *jgit* library, therefore avoiding checking out the entire project on disk.

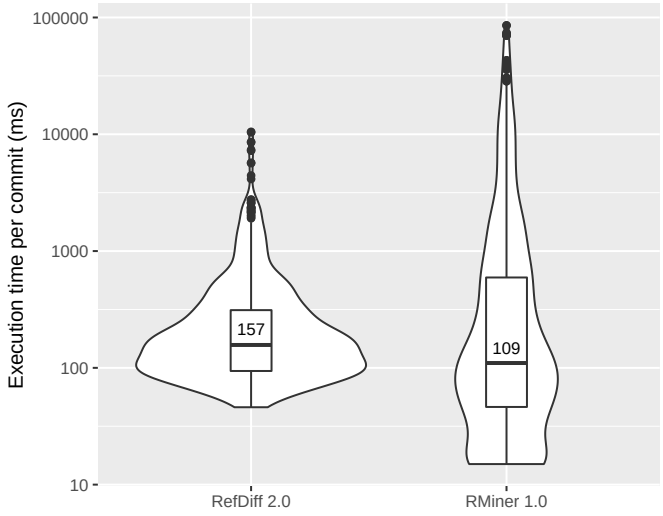


Fig. 7. Violin plot of execution time per commit in log-10 scale

4.4 Threats to Validity

There are at least two threats to the validity of the evaluation with Java projects. The first one is the subjectivity inherent to the manual classification of the reported refactorings as true/false positives, which directly impact the computed precision. Different validators may have a different interpretation of the code change under analysis, which is demonstrated by the fact that in 34% of the cases the validators initially disagreed. We mitigated this threat by having each refactoring assessed by two validators independently. Second, as discussed in Section 4.1, our oracle can not be granted as complete, i.e., there might exist refactorings in the analyzed revisions that are not detected by any of the tools. Thus, the actual recall might be lower than the computed recall. Unfortunately, it is not feasible to assure the completeness of an oracle at this scale with manual inspection. A single commit usually contains hundreds of changed lines of code, making such task extremely time consuming and error prone. Nevertheless, the computed recall serves the purpose of comparison between tools, and it should also be a fair approximation of actual recall, as our oracle is based on refactorings found by three tools: RMiner 1.0, RefDiff 1.0, and RefDiff 2.0.

6. We repeated the experiment three times and discarded the times collected in the first run, which was considered as a warm up.

5 Evaluation with JavaScript and C

Besides the Java evaluation, we also evaluated precision and recall of RefDiff in JavaScript and C. Unfortunately, we did not find a dataset with detailed information about real refactorings performed in these languages that we could use as an oracle. Therefore, we had to adopt a different strategy. To evaluate precision, we manually validated the refactorings detected by RefDiff in a set of GitHub repositories, in both languages (Section 5.1). Then, to evaluate recall, we searched for documented refactoring operations in commit messages of the same set of repositories (Section 5.2). After that, in Section 5.3, we report the precision and recall achieved by RefDiff. We are not aware of any other tool for detecting refactorings in these languages. Therefore, in this second evaluation, it was not possible to compare RefDiff's results with competitor tools.

5.1 Evaluation Design: Precision

To compute RefDiff's precision when detecting refactorings in JavaScript and C, we followed these steps:

1) We selected the 20 most popular GitHub repositories of each language. For this, we queried the GitHub API for repositories, sorting by stars count—which is a reliable indicator of popularity in GitHub [28], [29] —and filtering by programming language. The resulting list of repositories was manually inspected to discard the ones that are not actual software projects, e.g., tutorials or code samples. Then, we forked each selected repository, to preserve their version histories from future changes pushed to the original project. Table 5 shows the name, short description, and number of commits of each selected repository, both for JavaScript and C.
2) We ran RefDiff in the version history of each repository. To select the commits, we navigate the commit graph backwards, starting from the most recent commit in the master branch. We also discarded merge commits, i.e., commits which have two predecessors. The rationale is that comparing a merge commit with their predecessors results in duplicated reports of refactorings applied in the commits

TABLE 5
JavaScript and C repositories used in the evaluation

Repository	Description	Commits
react	A declarative, efficient, and flexible JavaScript library for building user interfaces.	10,964
vue	Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web.	3,014
d3	Bring data to life with SVG, Canvas and HTML.	4,148
react-native	A framework for building native apps with React.	16,875
angular.js	AngularJS - HTML enhanced for web apps.	8,963
create-react-app	Set up a modern web app by running one command.	2,233
jquery	A fast, small, and feature-rich JavaScript library.	6,403
atom	The hackable text editor.	36,752
axios	Promise based HTTP client for the browser and node.js.	847
three.js	JavaScript 3D library.	27,762
socket.io	Realtime application framework (Node.JS server).	1,715
redux	Predictable state container for JavaScript apps.	2,819
webpack	A bundler for javascript and friends.	7,852
Semantic-UI	Semantic is a UI component framework based around useful principles from natural language.	6,684
reveal.js	The HTML Presentation Framework.	2,341
meteor	Meteor, the JavaScript App Platform.	21,966
express	Fast, unopinionated, minimalist web framework for node.	5,555
material-ui	React components for faster and easier web development.	9,449
Chart.js	Simple HTML5 Charts using the canvas tag.	2,739
linux	Linux kernel source tree.	839,761
netdata	Real-time performance monitoring, done right!	8,338
redis	Redis is an in-memory database that persists on disk.	8,158
git	Git is a free and open-source distributed version control system.	55,723
ijkplayer	Android/iOS video player based on FFmpeg n3.4, with MediaCodec, VideoToolbox support.	2,584
php-src	The PHP Interpreter.	112,847
wrk	Modern HTTP benchmarking tool.	76
the_silver_searcher	A code-searching tool similar to ack, but faster.	2,016
emscripten	Emscripten: An LLVM-to-Web Compiler.	19,468
vim	The ubiquitous text editor.	9,875
jq	Command-line JSON processor.	1,287
FFmpeg	A complete, cross-platform solution to record, convert and stream audio and video.	93,898
tmux	A terminal multiplexer: it enables a number of terminals to controlled from a single screen.	7,618
nuklear	A single-header ANSI C gui library.	1,646
obs-studio	Free and open-source software for live streaming and screen recording.	6,727
libuv	Cross-platform asynchronous I/O.	4,319
swoole-src	Coroutine-based concurrency library for PHP (like Golang).	9,938
curl	A command line tool and library for transferring data with URL syntax.	24,339
toxcore	The future of online communications.	3,771
darknet	Convolutional Neural Networks.	436

prior to the merge operation. Moreover, to avoid over-representing projects with longer histories, we established a limit of 500 commits per repository. For each selected commit, we compared its source code with its predecessor using RefDiff, to detect refactoring operations.

3) Given the list of refactorings detected by RefDiff, we randomly selected 10 instances of each refactoring type to manually assess whether they correspond to actual refactorings (true positives), or incorrect reports (false positives). When applying the random selection, we enforced the constraint that we should not select two refactoring instances performed in the same commit. In this way, we avoid selecting similar refactorings which were performed in batch, e.g., multiple classes or functions moved together. To confirm each refactoring operation, one of the authors manually inspected the diff of the code changes in the corresponding commit.

After following the three steps, we compute precision as $P = TP/(TP + FP)$, where TP is the number of true positives and FP is the number of false positives.

5.2 Evaluation Design: Recall

To compute RefDiff's recall when detecting refactorings in JavaScript and C, we followed three steps:

1) We used GitHub API to find refactorings documented in commits from the repositories selected for evaluating precision (Section 5.1). Such queries consist in searching for keywords denoting a particular type of refactoring in the commit message, as described in Table 6. For example, when looking for *Rename Function* refactoring instances, we built

a query that looks for commits which contain the keywords *rename* and *function* in their messages, among other combinations.

2) Given the list of results of a query, one of the authors manually inspected each item to assess whether it really contains a refactoring. He started by analyzing the commit message. In many cases, the keywords are found in the text, but they are not referring to a refactoring operation. For example, one of the messages was: *"The routeToRegExp() function, introduced by 840b5f0, could not extract path params if the path contained question mark or hash."* This message contains the keywords *extract* and *function*, but clearly does not describe an *Extract Function* refactoring. In these situations, he discarded the commit with no further analysis. When the commit message described a refactoring, he checked the code diff to confirm it. Each confirmed refactoring was recorded in a normalized textual format compatible with the output of RefDiff. Inspecting the code diff is also necessary to locate the code elements involved in the operation. For example, the message *"Extract commit phase passes into separate functions"* documents a *Extract Function* refactoring, but does not specify the name of the extracted functions. He repeated this procedure until he found 10 instances of each refactoring type or when there were no more results to inspect. We found less than 10 instances when looking for *Move and Rename File*, *Move and Rename Function*, and *Inline Function* for JavaScript. Additionally, although modern JavaScript contains classes, we did not find documented refactorings instances of *Move and/or Rename Class*.
3) We ran RefDiff in the commits that contain documented and manually-validated refactorings to assess whether they are reported (true positives) or missed (false negatives).

After following these steps, we compute recall as $R = TP/(TP + FN)$, where TP is the number of true positives and FN is the number of false negatives.

TABLE 6
Search queries for each refactoring type

Change Signature	add parameter, remove parameter, add argument
Move/Rename File	move file, rename file, move folder, move rename file, move rename
Move/Rename Function	move function, rename function, move and rename
Extract Function	extract, duplicate, extract function, factor out
Inline Function	inline, inline into, remove indirection, indirect functions, remove wrapper

5.3 Results for JavaScript and C

Table 7 shows the precision and recall results for JavaScript. The overall precision is 91%. There are three refactorings with precision of 80%: *Rename Function*, *Move and Rename Function*, and *Inline Function*. For the remaining refactoring types, RefDiff has a precision of 90% (two refactoring types) or a precision of 100% (five refactoring types). Table 7 also shows the recall results, which reach 88% when all refactoring types are considered together. *Inline function* has the lowest recall (40%); however, our dataset has only five instances of this operation. There are three refactoring types with recall of 100%: *Move File*, *Move Function*, *Move and Rename File*. For the other ones, recall ranges between 80% and 90%.

TABLE 7
JavaScript precision and recall results

Refactoring Type	#	Precision	#	Recall
Move File	10	1.00	10	1.00
Move Class	2	1.00	0	
Move Function	10	0.90	10	1.00
Rename File	10	1.00	10	0.80
Rename Class	5	1.00	0	
Rename Function	10	0.80	10	0.90
Move and Rename File	10	1.00	3	1.00
Move and Rename Function	10	0.80	7	0.86
Extract Function	10	0.90	10	0.90
Inline Function	10	0.80	5	0.40
Total	87	0.91	65	0.88

Table 8 shows the precision and recall results for C. The overall precision is 88%. *Inline Function* is the refatoring for which precision is lower (50%). Besides, there are two refactorings with precision of 80%: *Move Function* and *Move and Rename Function*. For the remaining refactoring types, RefDiff has a precision of 90% (one refactoring type) or a precision of 100% (four refactoring types). We did not find any instance of *Move and Rename File*, thus we could not compute precision for this refactoring type. Table 8 also shows the recall results, which is 91% overall. *Extract Function* (70%) and *Move Function* (80%) are the ones with lowest recall. For the remaining refactoring types, RefDiff has a recall of 90% (three refactoring types) or a recall of 100% (four refactoring types).

TABLE 8
C precision and recall results

Refactoring Type	#	Precision	#	Recall
Change Signature	10	1.00	10	0.90
Move File	10	1.00	10	1.00
Move Function	10	0.80	10	0.80
Rename File	10	1.00	10	1.00
Rename Function	10	0.90	10	1.00
Move and Rename File	0		10	1.00
Move and Rename Function	10	0.80	10	0.90
Extract Function	10	1.00	10	0.70
Inline Function	10	0.50	10	0.90
Total	80	0.88	90	0.91

5.4 Threats to Validity

One threat to validity of the evaluation with JavaScript and C projects is its smaller scale. For example, we computed precision for JavaScript using 87 refactorings, and recall using 65 refactorings, while the evaluation in Java used an oracle with 3,249 refactorings. Particularly, we restricted the analysis to 10 instances per refactoring type. First, we acknowledge this limit does not express the frequency of each refactoring in practice. Second, as a result of this decision, our evaluation dataset for JavaScript and C is not complete with respect to true positives. In fact, the evaluation with JavaScript and C is a complement to the evaluation with Java aiming to show that RefDiff 2.0 provides similar results when used to detect refactorings in other programming languages. Last, we should also note that, similarly to the evaluation with Java projects, the subjectivity inherent to the manual classification of the reported refactorings is also a threat to validity.

6 Challenges and Limitations

Low-level refactorings: RefDiff does not detect local refactorings, such as rename, extract or inline a local variable, because the syntactical structure of the source code within a CST node is not represented. While it is theoretically possible to extend the CST to include finer-grained code elements such as statements, local variables, and others, this would also make it harder to port RefDiff to other programming languages.

Generating call graphs: In our modular architecture, the generation of the CST, which includes information from a call graph and a type hierarchy graph, is delegated to a language-specific plugin. For languages such as Java, there are reliable parsers and static analyzers that aid in this task (e.g., Eclipse JDT). However, we acknowledge that generating precise call graphs for untyped languages, such as JavaScript, might be a challenging problem. Nevertheless, we provided evidences that our approach works well even when the information encoded in the CST is not completely precise. For example, in our JavaScript implementation—which contains only 689 lines of code in total—we used a simple strategy in which we assume a node n_1 uses n_2 if n_1 contains a function call with the same identifier as n_2 and both are defined in the same file. However, to detect a *Extract* relationship between n_1 and n_2, we need two other conditions: (i) n_2 should be a new method and (ii) the body of n_2 should be similar to the code removed from n_1 between revisions. In other words, an incorrect edge in the call graph only leads to an incorrect *Extract* relationship in the unlikely scenario in which a function n_2 is introduced, the content of such function is similar to code removed from n_1 and n_1 calls a function with the same identifier of n_2 after the change, but that function is not actually n_2. A similar reasoning applies to *Inline* relationships, which also depends on information from call graphs. In summary, although generating precise call graphs is non trivial for untyped languages, we argue that it is not needed in practice to achieve acceptable precision, specially in the light of the results of our evaluation using JavaScript systems (91% of precision).

JavaScript class syntax: Our JavaScript implementation only considers classes defined with the new ES6 syntax, i.e., classes emulated by functions definitions and prototype-based inheritance are just treated like regular functions when generating the CST.

Field-related refactorings: As our refactoring detection algorithm is centered around code similarity and fields do not have a body, we did not implement the detection of *Move/Pull Up/Push Down Field* in RefDiff 2.0. Unrestricted detection of *Move Field* based solely on fields' types and names is prone to find many false positives. However, we plan to add support for field-related refactorings in future work by using stricter detection rules, similarly to RMiner (e.g., requiring a dependency between their source and destination classes).

7 Conclusion

To the best of our knowledge, RefDiff 2.0 is the first refactoring detection approach that supports multiple programming languages. We made this possible with two main design decisions. First, our refactoring detection algorithm relies only on information encoded in CSTs, a data structure that represents the source code but abstracts the specificities of each programming language. Second, we compute code similarity at the level of the tokenized source code, using techniques from information retrieval. In summary, RefDiff is loosely coupled to the syntax of the target programming language, which makes it easier to extend it to other languages. Our evaluation using a dataset of real refactorings in Java showed that RefDiff's precision is 96.4% and recall is 80.4%. Although we were not able to surpass RMiner's precision of 98.8%, we argue that we achieved satisfactory results for a language-neutral approach. In one hand, specialized tools can use more advanced techniques to improve refactoring detection. On the other hand, the higher the coupling with the syntax of a particular language, the harder it becomes to port the approach to other programming languages. Last, our evaluation in JavasScript and C also showed promising results. RefDiff's precision and recall are respectively 91% and 88% for JavaScript, and 88% and 91% for C. These results show the viability our approach for languages other than Java. Thus, we claim that RefDiff 2.0 can pave the way for important advances in refactoring studies in JavaScript, C, and other languages in the future. Moreover, it can be employed in practical tasks, such as improving diff visualization, automatically documenting refactorings in commits, keeping track of the history of refactored code elements, and others.

Acknowledgments

Our research is supported by grants from FAPEMIG and CNPq.

References

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

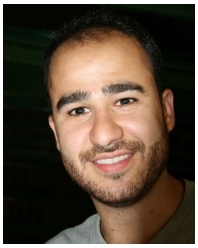
[2] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[3] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 132–146.
[4] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *20th Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 50:1–50:11.
[5] ——, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, July 2014.
[6] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *24th Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.
[7] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *27th European Conference on Object-Oriented Programming (ECOOP)*, 2013, pp. 552–576.
[8] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of API-level refactorings during software evolution," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 151–160.
[9] P. Weißgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *3rd Workshop on Mining Software Repositories (MSR)*, 2006, pp. 112–118.
[10] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *12th Conference on Source Code Analysis and Manipulation (SCAM)*, 2012, pp. 104–113.
[11] A. Hora, D. Silva, R. Robbes, and M. T. Valente, "Assessing the threat of untracked changes in software evolution," in *40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1102–1113.
[12] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.
[13] Z. Xing and E. Stroulia, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 951–952.
[14] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 1–11.
[15] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *40th International Conference on Software Engineering (ICSE)*, 2018, pp. 483–494.
[16] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 2018.
[17] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *5th Working Conference on Mining Software Repositories (MSR)*, 2008, pp. 35–38.
[18] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.
[19] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *ACM SIGPLAN Notices*, vol. 35, no. 10, 2000, pp. 166–177.
[20] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *20th European Conference on Object-Oriented Programming (ECOOP)*, 2006, pp. 404–428.
[21] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st International Conference on Automated Software Engineering (ASE)*, 2006, pp. 231–240.
[22] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.
[23] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *8th Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 371–372.
[24] Z. Xing and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," in *20th International Conference on Automated Software Engineering (ASE)*, 2005, pp. 54–65.
[25] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta, "On the impact of refactoring operations on code quality metrics," in *30th International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 456–460.
[26] G. Salton and M. J. McGill, *Introduction to modern information retrieval*. McGraw-Hill, 1986.
[27] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii, "Finding the jaccard median," in *21st Symposium on Discrete Algorithms (SODA)*, 2010, pp. 293–311.
[28] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 334–344.
[29] H. Silva and M. T. Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

Danilo Silva is a Ph.D. candidate in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he also received a M.Sc. in Computer Science. His research interests include software architecture and modularity, software maintenance and evolution, and refactoring. Contact him at danilofs@dcc.ufmg.br.

João Paulo Ribeiro da Silva is a bachelor in Computer Science, majored at the Federal University of Campina Grande (UFCG). He is a developer and team lead at Quimbik, Inc., working on front and back-end software for web systems. His main interest areas are software architecture and modularity, software maintenance and evolution, and web development. Contact him at joao@jpribeiro.com.br.

Gustavo Santos is an assistant professor in Software Engineering at the Federal University of Technology (UTFPR) in Dois Vizinhos, Brazil. He received his Ph.D. in Informatique from Université de Lille (France) in 2017 and stayed two years at University of São Paulo (USP) for post-doctoral research. His research interests include software maintenance and software quality. Contact him at gustavosantos@utfpr.edu.br or visit gustavojss.github.io.



Ricardo Terra received his Ph.D. degree in Computer Science from Federal University of Minas Gerais, Brazil (2013) with a 1-year internship at the University of Waterloo, Canada. Since 2014, he is an assistant professor in the Department of Computer Science at Federal University of Lavras, Brazil. His research interests include software architecture maintainability and evolvability. Contact him at terra@ufla.br, or visit www.dcc.ufla.br/~terra.

Marco Tulio Valente is an associate professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he also heads the Applied Software Engineering Research Group (ASERG). His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. Valente received a Ph.D. in Computer Science from the Federal University of Minas Gerais. He is a Researcher I-D of the Brazilian National Research Council (CNPq) and holds a Researcher from Minas Gerais State scholarship, from FAPEMIG. Contact him at mtov@dcc.ufmg.br, or visit www.dcc.ufmg.br/~mtov.