Support New Programming Language in RefDetect

Sandu-Victor Mintuș
s.mintus@student.utwente.nl
University of Twente
Enschede, The Netherlands

ABSTRACT

Refactoring plays a crucial role in software development. It represents the process of modifying and improving the structure of the code, without changing the behaviour of the software itself. There exist a large number of tools that can detect code refactorings, however, one major drawback of them is that they are language specific. RefDetect is a language-agnostic tool that uses a string-alignment algorithm to detect code refactoring which currently supports Java and C++ but can be extended to support any class-based, object-oriented programming language. The paper aims to describe such an extension for Kotlin and argue about the performance of this approach compared to the current state-of-the-art tool, namely KotlinRMiner.

KEYWORDS

RefDetect, KotlinRMiner, Code refactoring, Program Structure Interface

1 INTRODUCTION

Developers have widely used refactoring to improve the quality of software. It improves maintainability and scalability by removing duplicate code and reducing complexity. [14] presents a study on the impact of refactorings on software quality. The study has used a series of metrics to express the quality of the software, such as bad smells, cyclomatic complexity and depth of inheritance tree. It showed that, according to these metrics, refactorings significantly improve the quality of the software. Furthermore, software quality has an impact on the productivity of developers. A study from [5] states that higher software quality leads to fewer bugs, consequently reducing the time spent on troubleshooting and fixing errors, which developers can use to implement new features.

Detecting code refactorings has also become an important topic. It provides a good overview of how the software has evolved [17]. Likewise, it gives an insight into the developer's thought process and understands the reasoning behind certain decisions. Despite their benefits, refactorings can also create potential problems in the development process. For instance, they can introduce merge conflicts between different development branches [16] or erroneously trigger algorithms that check for induced bugs [9]. To mitigate these issues, several refactoring-aware tools have been developed, such as [21] for branch merging and [19] for detection of bugs induced in code. These tools rely on the identified performed refactorings, therefore there needs to be a good support in this regard. Furthermore, researchers who study code refactorings, like [8], need a reliable tool for extracting them so that they obtain significant results in their research.

There have been several proposed approaches to detect code refactorings. One of these tools is RefDetect[17]. This language-agnostic tool provides a novel approach to refactoring detection using a string alignment algorithm to determine changes between two program versions. It matches entities between the programs using their signature and relationship with other entities. This tool has proven to be effective for both Java and C++ programs.

Kotlin is a popular object-oriented programming language designed to be a less verbose version of Java. While Java remains the industry standard, Kotlin has grown considerably in the last few years, so in 2019, google announced it as the preferred language for Android development [2]. In a survey by JetBrains, 18 % of participants stated that they used Kotlin in the past year. Moreover, 8 % of participants stated there is a high chance they will learn Kotlin in 2023, the third most after Go and Rust.[13].

Considering the increasing popularity of the language, more research needs to be performed on detecting code refactorings in Kotlin. Currently, only one tool exists, namely, KotlinRMiner[1], based on RMiner 2.0. [24]. The tool has yet to be evaluated, as the authors are still working on creating a representative dataset of Kotlin refactorings. Therefore, there is no information about the performance of KotlinRMiner.

Since RefDetect can be extended to support any object-oriented language, we have decided to implement it for Kotlin, given its popularity, relevance and the lack of research in the area. Therefore, this paper aims to present a new tool for detecting refactorings in Kotlin by extending RefDetect and then comparing its performance to KotlinRMiner.

2 RESEARCH QUESTION

We have formulated two research questions to achieve the goal of the project.

(1) How can RefDetect be extended so that it also supports Kotlin?
(2) Does RefDetect perform better on Kotlin code refactorings than KotlinRMiner?

3 RELATED WORK

This section will cover research that has been performed in detecting code refactorings. Since this is not an innovative approach to this problem, we will mention existing tools that have developed

39th Twente Student Conference on IT July 7th,, July 7, 2023, Enschede, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

over the years, but focus on the main relevant tools, RefDetect, RMiner 2.0, and KotlinRMiner.

Over the years there have been several tools developed to detect code refactorings. They have used different approaches, such as low-code metrics defined as heuristics [10], a signature-based technique based on token-based code clone detection [26], text-based similarity metric [12], representing the code as UML models and then detecting differences between these models [27] or representing the code entities and their relations as predicate logic [20]. While all these approaches are valid and have their benefits, they have several drawbacks such as using commit logs, failing to detect edge cases and relying too much on user-provided code similarity thresholds.

RMiner 2.0 [24] is an improved version of RMiner [25]. It uses the Git unified diff format to detect lines that have been added or removed and uses a set of heuristics and rules to determine if these changes represent code refactorings. Therefore, it does not rely on user-provided code similarity thresholds which increases the tool's accuracy. Moreover, it has a fallback mechanism that allows it to handle unparsable programs, a feature lacking from other tools. The authors have reported a 99.6% accuracy and a 94% recall, which is a good indicator of the tool's performance. RMiner 2.0 has also served as a basis for other tools, such as JsDiffer [22] and PYREF [4] which can detect refactorings in Javascript and Python respectively, thus showcasing its widespread usage and influence. However, there are some drawbacks to the approach used RMiner 2.0 that affect the refactoring process. For instance, the tool only analyzes files modified between 2 commits, so it lacks additional context for the project it analyzes. This can cause a mislabeling of the type of the refactoring. For instance, pull-up refactorings can be labelled as a move, if the entity is pulled multiple levels up [24]. Moreover, the tool relies heavily on matching entities based on their name and type but does not consider the relationships between them. [17] shows that this causes RMiner 2.0 to detect incorrectly a Move Field refactoring when a field is deleted from a class, and a new similar field is created in another class. Finally, a drawback is that the tool is language-specific for Java, so it has limited applicability to the field.

KotlinRMiner [1] is an extension of [24] for detecting code refactorings in Kotlin developed by JetBrains research. The sole mention of KolinRMiner is in [15], a paper that describes an Intellij plugin for detecting and representing code changes for Java and Kotlin. It relies on KotlinRMiner for detecting the refactorings in Kotlin, however, it does not state any results about the performance of KotlinRMiner in terms of precision and recall, and the authors do not describe how they implemented the tool. However, since KotlinRMiner is an extension [24], we can assume that the algorithm used for detecting refactorings is the same.

RefDetect [17] is code refactoring detecting that uses a differential algorithm to determine changes between two program versions. For this, it transforms the code into a string representation based on the following entities: Class (C), interface (I), generalisation relationship (G), attribute (A), method (M), method parameter (P), and a call connection between two classes as (R) and uses a string alignment algorithm called FOGSAA [7] to compare the strings and detect the changes. Then, it uses the signature and relationships of different entities to match them between the versions. While it still uses similarity thresholds to determine possible matches between the entities, RefDetect uses a two-step algorithm to determine the refactorings. In the first round, the algorithm detects refactorings primarily based on the similarity of names. Therefore it mainly detects classes, methods, or fields that have been renamed, but whose relationships were barely changed. In the second step, the algorithm focuses on relationships between entities rather than on name similarity. Moreover, refactorings detected in the first step are also used in the detection process and the threshold value is also increased. This step reduces the number of false negatives and false positives detected by the algorithm. A visual representation of the algorithm can be seen in figure 1. RefDetect addresses some of the limitations encountered RMiner 2.0. First of all, RefDetect is designed to be language-agnostic, which increases its applicability to other languages. Moreover, it uses relationships between entities during the refactoring process, which increases the reliability of the detected refactorings.

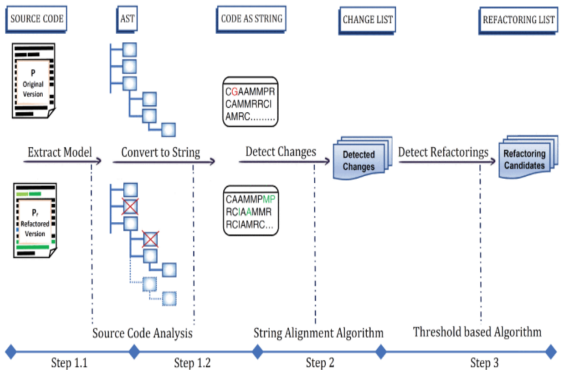


Figure 1: RefDetect algorithm

4 METHODOLOGIES

Since RefDetect is language-specific only in the parsing phase, the only step required to support Kotlin is to find a parser for Kotlin code and use it to extract relevant information from the source code.

4.1 Parsing of Kotlin Code

The first step of this research was to find a suitable parser for Kotlin to generate the AST(Abstract Syntax Tree). We have considered several tools for this, such as kotlinx.ast [11], Kastree [6] or even using Antlr4 [23] to generate the parse tree from the Kotlin grammar. The first one turned out to be still at an early stage of development, so it is not reliable enough. Likewise, Kastree has not been updated for 4 years and is no longer maintained. Finally, using the Antlr4-generated parser is a cumbersome process since it is necessary to traverse the tree manually to perform operations on the nodes. Ultimately, we have decided to use PSI (Program Structure Interface), the layer in IntelliJ responsible for parsing files. The PSI can parse

the Kotlin files and manipulate the entities. Moreover, it is also used by KotlinRMiner, so its efficiency has been verified in practice.

To generate the PSI representation of the Kotlin files, we have used the Kotlin Compiler Embedabble dependency. It allowed us to access the Kotlin Compiler API, which contained the necessary tools for creating and accessing the PSI files.

4.2 Generation of Source Information

RefDetect uses a particular data structure called Source Information which contains all relevant information about the source code. This data structure includes all the classes of the projects as well as the relationship among these classes. This source information serves as input for the String alignment algorithm, which is the next step of the detection process used by RefDetect. Therefore, the end goal of the parsing step is to generate the Source Information for the Kotlin classes. PSI was a good tool for processing basic information about the Kotlin classes. However, we encountered two significant limitations of the PSI that slowed down the parsing part considerably. First, PSI does not contain information about the inferred types of variables. Since type inference is a widespread practice in Kotlin, finding a workaround to this problem was crucial. Likewise, PSI does not provide the binding context for name references used in the project. Binding context is essential because Source Information requires precise information about the parsed code. In the following two subsections, we describe how we overcome these difficulties.

4.2.1 Type Inference.

Type inference is a feature of the Kotlin compiler that allows for types to be deduced at compile time so that they need not be explicitly declared in the code. PSI does not have any information about inferred types since compilation happens after the PSI representation of the file is generated. To tackle the issue, we have considered several possibilities.

First, we considered ignoring type inference as a feature altogether and assuming for the scope of this research to work with Kotlin which uses explicit type systems. This was the easiest option for us to choose; however, it would have diminished the relevance of this research because type inference is one of the main features of Kotlin.

Likewise, we thought of using a particular type called "Untyped". This is the approach that is being used by KotlinRMiner when dealing with type inference. Ultimately, we decided against it because the string alignment algorithm relies heavily on the types of entities when it compares the different versions of the program. Thus, setting the type of different entities to "Untyped" would make the algorithm less reliable, so the tool's accuracy would suffer.

We have also tried manually creating the binding context of the entire project. If the binding context is set correctly, then it can be used to infer the types of the variable manually. The main problem that we encountered with this approach is that there needs to be more online support that we could find that would help us with this approach. Moreover, even though the Kotlin compiler is open source, it does not have any documentation that states how it can be used. We have also researched the source code but have yet to find any valuable information for us in this regard.

In the end, we have decided to make our implementation of type inference. This was the most reliable way to handle this problem, even if it required considerably more effort from our side. First, we implemented type inference for fields by evaluating the expressions used to initialize them. We have identified the possible kinds of expressions that can be used. These include name references, method calls, constructors, constants and binary expressions. For name references, we simply return the type of that reference. We perform the type inference process on this name reference if the type is not assigned. For method calls, we return the type of the method or perform type inference on the method. We use the Jexl3 library from org.apache to evaluate constants and binary expressions.commons [3]. This library evaluates expressions written as strings and returns the evaluation result. We extract the class of the result and return it as the inferred type of the expression. A limitation of this library is that it requires the expression to have defined values. However, a binary expression may contain a reference to a name. In that case, we substitute that referenced name with a concrete value of that type. We have implemented this for the basic types of Kotlin. Next, we implemented type inference for the return type methods. For this, we evaluate the return expression of the method using the same approach described above. If the method does not contain an explicit return expression, we infer that the type of the method is void.

To illustrate the process, we will present the inference process based on the example from figure 2. In the method main of class A, we have two fields, c and r. Field c is initialized using a constructor; therefore the inferred type will be the constructor's class, in this case, Calculator. Next, field r is initialized using field c, which calls method sum. We had already inferred that the type of c is Calculator; therefore the type of field r is the return type of the method sum from the class Calculator. The problem is that the class Calculator can have multiple methods with the name sum but with different signatures. To mitigate this, we also infer the type of arguments used to call sum and use the entire signature to find the correct function. In this case, the first argument is a call from field c to method min. The call's arguments are constants, so using the type inference mechanism described above, we infer that the type of both arguments is Double. Now, we can deduce the method's signature and use it to find the correct method in the class Calculator and return its type. In this case, the return type of method min(Double, Double) from the class calculator is Double, so we infer that the type of the first argument of the method sum is Double. The second argument is a Double constant, so we return its type. Using the same approach for method min(Double, Double), we infer that the type of field r is the type of the return type of method sum(Double, Double) from the class Calculator. We can notice that method sum does not have an explicit return type. In this case, the return expression of the method needs to be evaluated. The return expression of method sum is the binary expression "x+y", where x and y are both fields of type double. To evaluate this expression, both name references will be replaced by a concrete instance of that type. Name references of type Double are replaced by "1.0", so the evaluated expression is "1.0 + 1.0". The result of the evaluation is "2.0", which has type Double; therefore the return expression has type Double. Since the return expression has the type Double, we can conclude that the method sum(Double, Double) has the return type Double. Subsequently, field r also has type Double.

4.2.2 Binding Context.

```
class A {
    new *
    companion object {
        new *
        fun main(args: Array<String?>?) {
            val c = Calculator()
            val r = c.sum(c.min(2.3, 3.0), 1.8)
            var d = c + 2.0
            print(r)
        }

        new *
        private fun print(res: Double) {
            System.out.printf("%.2f", res)
        }
    }
}
```

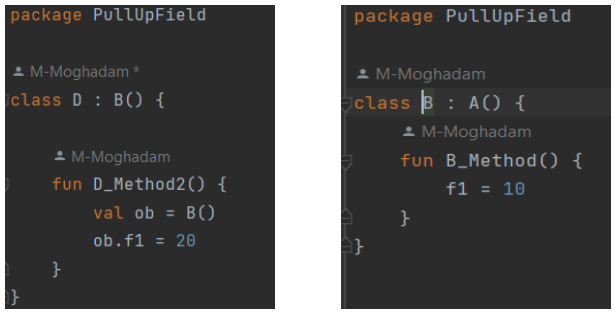
(a) Property Type Inference

```
M-Moghadam *
internal class Calculator {
    M-Moghadam *
    fun sum(x: Double, y: Double) {
        return x + y
    }

    M-Moghadam *
    fun min(x: Double, y: Double) : Double {
        return if (x < y) x else y
    }
}
```

(b) Method Type Inference

Figure 2: Example of Type Inference



```
package PullUpField

M-Moghadam *
class D : B() {

    M-Moghadam
    fun D_Method2() {
        val ob = B()
        ob.f1 = 20
    }
}
```

(a) Class D

```
package PullUpField

M-Moghadam
class B : A() {
    M-Moghadam
    fun B_Method() {
        f1 = 10
    }
}
```

(b) Class B

```
package PullUpField

M-Moghadam
class A {
    var f1 = 0
}
```

(c) Class A

Figure 3: Example of Inheritance

We have mentioned binding context in the previous section as a possible solution to the problem of type inference. However, binding context was essential for other aspects of the project as well. One of the most important features of object-oriented programming languages is inheritance. Without a binding context, it is not clear where a particular entity has been declared. In the example of figure 3 we have field f1 declared in class A and entity ob of type B which accesses this field. Without looking at classes A and B, it would seem that f1 is declared in class B; however, class B inherits this field from A. To solve this issue, we check all possible classes from where a class could have inherited an entity until we get a match. This includes outer classes, parent classes, and parent interfaces (in this order of search). We performed the matching of entities based on their signature. In case the entity is missing the type, we perform type inference as described above

4.3 Experiment

To test the tool we used a set of small Java projects previously used to test RefDetect. These projects contain a set of 390 true applied refactorings that have already been extracted. We had considered using the set of projects from [18], however, we have decided against it because of time limitations.

For answering RQ1, we have designed JUnit test cases that compared the generated source information of the two parsers. If these data structures turn out to be the same, then it is safe to say that the Kotlin parser can be used by the RefDetect algorithm. This is because the parsing phase is the only language-specific part of the algorithm. Therefore, if the algorithm receives the same source information as the input, it will yield the same results for the refactoring detection.

For answering RQ2 we executed RefDetect and KotlinRMiner on the projects mentioned above and compared their performance. Since the datasets are relatively small, this validation of the results was done manually. For detecting refactorings, KotlinRMiner requires the project to be linked to a remote repository. For this, we have created a repository and we pushed each project to a separate branch in this repository. Then, we executed the tool on each branch and documented the results. We decided to use precision, recall and f-score as metrics for evaluation. The formulae for these metrics can be seen below.

$$Precision = \frac{\#\text{ of correct refactorings}}{\#\text{ of recommended refactorings}} \quad (1)$$

$$Recall = \frac{\#\text{ of correct refactorings}}{\#\text{ of true refactorings}} \quad (2)$$

$$\text{f-score} = \frac{2 * Precision * Recall}{Precision + Recall} \quad (3)$$

5 RESULTS

5.1 RQ1

The JUnit tests showed that the Java and Kotlin parsers generate almost identical data structures of Source Information. Both parsers yield the same result for generalization relationships, association relationships of classes defined within the project, and the string representation of the classes. The differences occur when entities have relationships with classes defined outside the project's scope. In the scope of this research, these differences were minimal and not critical to the refactoring process, so in the end, we considered the generated Source Information to be the same for both parsers.

5.2 RQ2

The results for RQ2 can be seen in the table 1. We can see that RefDetect achieved high results in terms of both Precision (98%) and Recall (95%). On the other hand, KotlinRMiner achieved good results for Precision(96%) but mediocre results for Recall (56%). The f-score of RefDetect is also considerably higher than that of KotlinRMiner(0.95 and 0.72). The poor performance of KotlinRMiner was surprising, considering that RMiner2.0 achieved very high results for both Precision(99.6 %) and Recall(94Z%). Moreover, KotlinRMiner has failed to detect certain types of refactorings altogether, such as Rename Field refactorings, even though the authors claim that these refactorings can be detected. Thus, on this set of projects,

Table 1: Performance Metrics

Method	Precision	Recall	F1-score
RefDetect	0.98	0.93	0.95
KotlinRMiner	0.96	0.56	0.72

KotlinRMiner has performed better than KotlinRMiner in precision, Recall and f-score.

6 LIMITATIONS

As mentioned, we used PSI to extract information about the source code. We have already discussed how we tackled several limitations of PSI. However, these turned out to be very time-consuming. We had to consider many edge cases without reliable access to the project's binding context. This has caused a considerable delay in the research process, as most of these limitations were discovered at a late stage of the parsing phase. Moreover, considering the duration of the research, which is nine weeks (including two weeks for the proposal), we have yet to solve all the existing issues. One crucial problem that persists concerns classes defined outside the project's scope. Without a binding context, it is impossible to obtain additional information about these classes, information which is necessary for the generation of the Source Information. For the set of small projects we have used, this did not significantly affect the tool's performance. However, for larger projects with more complex statements, this can lead to potential problems in the detection process. Therefore, we cannot give a definite answer regarding the tool's scalability for large projects.

7 CONCLUSION

In this paper, we have described a Kotlin extension for the code refactoring detection tool RefDetect. We have shown that we can use PSI to extract source information from Kotlin code and that for the small set of projects that we used, the Source Information extracted by both Java and Kotlin parsers is almost the same, with minor differences related to the limitations of PSI.

We have also identified several limitations of using PSI as the parsing tool for the Kotlin code. These limitations include a lack of type inference and support for Binding Context, which affect the scalability of this tool for larger projects.

Likewise, we have shown that RefDetect performs better than KotlinR in terms of precision(98% and 96%), recall (93% and 56%) and f-score(0.95 and 0.92). Moreover, KolinRMiner failed to detect certain types of refactorings that it claimed, so it needs further validation of its performance.

8 FUTURE WORK

A possible direction for further research is to try other tools for generating the source information of the code. PSI turned out to be too limited for extracting complex information about the source code, thus other tools could provide more reliable information in this regard.

Furthermore, it is essential to check at an early stage of the parsing process that the tool used for extracting source code information can support type inference and binding context. They are essential for correctly analyzing Kotlin source code. If this is not the case, then it would be better to consider alternatives, because the lack of support for them drastically increases the complexity of the research, while reducing its reliability.

Likewise, the test set of projects that were used to test the tool is small, so a possible extension point is to use the dataset of projects provided by [18] to test the performance of the tool.

Finally, KotlinRMiner has not been analyzed in depth yet and, as we showed in our experiment, it still fails to detect a lot of refactorings. Therefore, further research needs to be done into the performance of KotlinRMiner and the validation of its claimed capabilities.

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Iman Hemati Moghadam of the EEMCS department for his input and help in this research, especially in the final stages of the project.

REFERENCES

[1] Zarina Kurbatova. 2021. kotlinRMiner. GitHub repository. https://github.com/JetBrains-Research/kotlinRMiner
[2] Android Developers. 2023. Kotlin First. https://developer.android.com/kotlin/first Accessed: 2023-07-02.
[3] Apache Commons. 2023. Apache Commons JEXL. https://commons.apache.org/proper/commons-jexl/
[4] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PYREF: Refactoring Detection in Python Projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 136–141. https://doi.org/10.1109/SCAM52516.2021.00025
[5] V. R. Basili and B. Boehm. 2001. Software Defect Reduction Top 10 List. *Computer* 34, 01 (jan 2001), 135–137. https://doi.org/10.1109/2.962984
[6] Chad Retz. 2019. kastree. GitHub repository. https://github.com/cretz/kastree Accessed: 25 June 2023.
[7] Angana Chakraborty and Sanghamitra Bandyopadhyay. 2013. FOGSAA: Fast optimal global sequence alignment algorithm. *Scientific reports* 3 (04 2013), 1746. https://doi.org/10.1038/srep01746
[8] Eunjong Choi, Norihiro Yoshida, and Katsuro Inoue. 2014. An Investigation into the Characteristics of Merged Code Clones during Software Evolution. *IEICE Transactions on Information and Systems* E97.D (05 2014), 1244–1253. https://doi.org/10.1587/transinf.E97.D.1244
[9] Daniel Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2016. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* PP (10 2016), 1–1. https://doi.org/10.1109/TSE.2016.2616306
[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding refactorings via change metrics. *ACM SIGPLAN Notices* 35, 166–177. https://doi.org/10.1145/353171.353183
[11] Dennis Rieks. 2022. kotlinx.ast. GitHub repository. https://github.com/kotlinx/ast
[12] Danny Dig, Ralph Johnson, Frank Tip, Oege Moor, Jan Becicka, William Griswold, and Markus Keller. 2007. Refactoring Tools. 193–202. https://doi.org/10.1007/978-3-540-78195-0_19
[13] JetBrains. 2023. https://developer.android.com/kotlin/first. https://www.jetbrains.com/lp/devecosystem-2022 Accessed: 2023-07-02.
[14] Amandeep Kaur and Manpreet Kaur. 2016. Analysis of Code Refactoring Impact on Software Quality. *MATEC Web of Conferences*, Article 02012 (May 2016), 6 pages. https://doi.org/10.1051/matecconf/20165702012
[15] Zarina Kurbatova, Vladimir Kovalenko, Ioana Savu, Bob Brockbernd, Dan Andreescu, Matei Anton, Roman Venediktov, Elena Tikhomirova, and Timofey Bryksin. 2021. Refactorinsight: Enhancing ide representation of changes in git with refactorings information. *arXiv preprint arXiv:2108.11202* (2021).
[16] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. 151–162. https://doi.org/10.1109/SANER.2019.8668012
[17] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. 2021. RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment. *IEEE Access* 9 (2021), 86698–86727. https://doi.org/10.1109/ACCESS.2021.3086689

[18] N. Tsantalis, A. Ketkar and D. Dig. 2023. Refactoring Oracle. Concordia University. http://refactoring.encs.concordia.ca/oracle/ Accessed: 25 June 2023.

[19] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2019. Revisiting and Improving SZZ Implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12. https://doi.org/10.1109/ESEM.2019.8870178

[20] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[21] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, Zhi Jin, and Qianxiang Wang. 2019. IntelliMerge: A refactoring-aware software merging technique. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28. https://doi.org/10.1145/3360596

[22] Mosabbir Khan Shiblu. 2022. *JsDiffer: Refactoring Detection in JavaScript.* Master's thesis. Concordia University, ontréal, Québec, Canada.

[23] Terence Parr, Sam Harwell,Eric Vergnaud,Peter Boyer,Mike Lischke,Dan McLaughlin,David Sisson,Janyou,Ewan Mellor, Hanzhou Shi,Ben Hamilton,Marcos Passos,Lingyu Li,Ivan Kochurkin ,Justin King,Ken Domino and Jim Idle. 2021. Antlr4. GitHub repository. https://github.com/antlr/antlr4

[24] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[25] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

[26] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. 231–240. https://doi.org/10.1109/ASE.2006.41

[27] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study. In *2006 22nd IEEE International Conference on Software Maintenance*. 458–468. https://doi.org/10.1109/ICSM.2006.52