PyRef: Refactoring Detection in Python Projects

Hassan Atwi*, Bin Lin*, Nikolaos Tsantalis†, Yutaro Kashiwa‡
Yasutaka Kamei‡, Naoyasu Ubayashi‡, Gabriele Bavota*, Michele Lanza*
*Software Institute – USI, Lugano, Switzerland — †Concordia University, Canada — ‡Kyushu University, Japan

***Abstract*—Refactoring, the process of improving the internal code structure of a software system without altering its external behavior, is widely applied during software development. Understanding how developers refactor source code can help gain better understanding of the software development process and the relationship between various versions of a system. Refactoring detection tools have been developed for many popular programming languages, such as Java (*e.g.,* RefactoringMiner and Ref-Finder) but, quite surprisingly, this is not the case for Python, a widely used programming language.**

Inspired by Refactoring Miner, we present PyRef, a tool that automatically detects method-level refactoring operations in Python projects. We evaluated PyRef against a manually built oracle and compared it with a Python-adapted RefactoringMiner, which converts Python program to Java and detects refactoring operations with Refactoring Miner. Our results indicate that PyRef can achieve satisfactory precision and detect more refactorings than the current state-of-the-art.

***Index Terms*—refactoring detection, Python, software maintenance**

I. Introduction

Refactoring, the process of improving the internal structure of a software system without changing its external behavior [1], has received significant attention by the software engineering research community. Understanding how refactoring is applied in software systems can help to gain insights into software maintenance and evolution, learning good software design practices and improve code comprehension. However, detecting refactoring is not a trivial task due to the fact that developers rarely document the refactoring operations they perform [2]. Besides, refactoring operations are often performed together with –or as a consequence of– other changes [3], which makes it even harder to distill them out of tangled code changes.

Several refactoring detection tools such as RefactoringMiner [4] and RefDiff [5] have been proposed to mine refactoring operations from software projects. Most of these tools mainly focus on the Java programming language [4], [6], [7], while RefDiff [5] also detects refactoring for projects written in JavaScript and C. None of these tools can be used to extract refactorings performed in Python. Given the fact that Python is currently one of the most popular programming languages[1], a refactoring detection tool specifically designed for Python might unlock many new research opportunities. For example, Python is the dominant language in the fields of scientific computing, data science, and machine learning [8]. Therefore, detecting refactoring in Python can allow to gain specific insights in these domains.

Dilhara and Dig [9] have taken the first step to address this issue and developed Python-adapted RefactoringMiner[2], which converts Python programs into Java and uses RefactoringMiner [4] to detect refactorings. However, there are considerable differences between these two languages, let alone the language grammar. For example, Python checks types at runtime while Java is a statically typed language. Moreover, Java is class-based and object-oriented, while Python projects can also follow other programming styles such as functional and imperative programming.

Inspired by RefactoringMiner [4], we present PyRef, a tool that automatically detects mainly method-level refactoring operations from Python projects. To evaluate the performance of PyRef, we ran it on three real-world Python projects, and manually validated the refactoring detection. We also compared PyRef with the only publicly available refactoring detection tool for Python, namely Python-adapted Refactoring Miner. On average, PyRef achieves a precision of 89.6% and a recall of 76.1%, which are both higher than the current state-of-the-art. This results show the potential of PyRef for refactoring detection in Python projects.

The remainder of the paper is structured as follows. Section II introduces current refactoring detection tools. The detailed techniques behind PyRef are described in Section III. Section IV reports the design and results of the study we performed to assess the performance of PyRef. The limitations of our tool are discussed in Section V. Finally, Section VI concludes the paper.

II. Related Work

The last two decades have seen a rapid growth of refactoring detection tools.

RefactoringCrawler, developed by Dig *et al.* [7], is one of these modern tools for refactoring detection. It uses a fast syntactic analysis based on Shingles encoding, a technique from the information retrieval field, to detect refactoring candidates. After locating the possible refactorings, it adopts a more expensive semantic analysis to refine the results. The performance of this tool was tested on three different projects (*i.e.,* EclipseUI, Struts, and JHotDraw) with manually extracted refactoring operations documented in these projects.

[1]https://www.tiobe.com/tiobe-index/

[2]https://github.com/maldil/RefactoringMiner

JDEvAN (Java Design Evolution Analysis), a tool developed by Xing and Stroulia [10], detects refactorings by applying a set of predefined queries on the design changes retrieved by UMLDIFF [11]. The evaluation of this tool was done on two software systems, and all of the documented refactorings were detected.

Prete *et al.* proposed REF-FINDER [12], another refactoring detection tool, which converts the target source code into logic predicates that describe the structure and elements of the code. The tool uses the Tyruba logic programming engine [13] to infer concrete refactoring instances with the help of a set of logical queries which identify the patterns of refactoring operations among the structural differences. It can detect 63 types of complex refactoring operations. The tool was tested on manually collected refactorings and it showed that its overall precision and recall are 79% and 95%, respectively.

Developed by Silva and Valente [14], REFDIFF detects 13 refactoring types through static analysis and code similarity comparison. REFDIFF first parses two versions of source code and transforms them into high level models that represent the code elements; it then analyzes both models to detect the relationships between the identical entities. TF-IDF is then use to compute the similarity between the code elements, and the similarity threshold is determined through a calibration process on a set of randomly selected systems. The tool was evaluated with an oracle of refactorings applied by qualified students. Silva *et al.* [5] extended the tool to other two languages (*i.e.,* C and JavaScript) with the same core approach. The precision of the tool for C and JavaScript projects was evaluated by manually validating the detected refactorings, while the recall was calculated based on refactorings documented in commit messages.

REFACTORINGMINER [4], developed by Tsantalis *et al.*, can detect over 80 different types of refactoring operation. REFACTORINGMINER differs from other tools as it does not require any similarity threshold. The tool detects the possible refactorings based on pre-defined detection rules. To evaluate REFACTORINGMINER, the authors used a dataset containing 7,226 true instances for 40 different refactoring types, which are validated by experts. The results indicate that it can achieve high average precision (99.6%) and recall (94%).

The most relevant work to ours is PYTHON-ADAPTED REFACTORING MINER developed by Dilhara and Dig [9]. This tool first converts Python code to Java and then feeds it into REFACTORINGMINER [4]. In theory, it supports the detection of all the refactoring types covered by REFACTORINGMINER. An early version of this tool has been released. However, as the authors are still improving the tool, currently no evaluation results have been published. While inspired by REFACTORINGMINER [4], unlike PYTHON-ADAPTED REFACTORING MINER, PYREF is a native refactoring detection tool for Python projects and it does not rely on program language translation or third party refactoring detectors.

III. PYREF

PYREF is a tool designed to mine refactoring operations in Python projects. As the tool is inspired by REFACTORINGMINER [4], the core approaches used in these tools are similar, with some adaptions due to the language differences. PYREF takes as input a Git repository of a Python project, and returns a list of refactoring operations performed in the project.

Currently PYREF supports the detection of 9 types of method-level refactoring operations: RENAME METHOD, ADD PARAMETER, REMOVE PARAMETER, CHANGE/RENAME PARAMETER, EXTRACT METHOD, INLINE METHOD, MOVE METHOD, PULL UP METHOD, and PUSH DOWN METHOD.[3]

PYREF follows five steps to detect refactoring operations: 1) extracting code changes, 2) modeling code elements, 3) matching code elements, 4) applying refactoring heuristics, and 5) sorting candidates.

A. Extracting Code Changes

PYREF iterates the commits in the history of a Python repository and takes two adjacent revisions (current commit and its parent) for refactoring detection. Merge commits are excluded to avoid redundant refactorings. Like REFACTORINGMINER [4], PYREF only analyzes changed files (*i.e.,* added, deleted, and modified) between two revisions to be executed more efficiently and reduce the chance of matching irrelevant code elements between revisions. Therefore, the first step of PYREF is to extract all the changed Python files.

B. Modeling Code Elements

PYREF then transforms the source code of each modified Python file into an abstract syntax tree (AST) using the built-in AST module.[4] It then converts Python AST instances into ANYTREE[5] instances to make it easier to navigate between AST nodes (*e.g.,* visiting the parent node). From each AST, PYREF extracts the needed code elements from each tree and represents them with a predefined model. We followed a similar information modeling approach as the one used by REFACTORINGMINER, including the following elements:

- **Module**: A module is defined by its name (same as the Python filename), contained classes and functions, and directly affiliated definitions and statements (*i.e.,* those not defined within a function or a method).
- **Class**: Classes may contain definitions and statements. For each class in a module, we assign it with its name, a list of inherited classes (if any), the defined fields, and methods.
- **Method/Function**: For each method/function, we associate it with the module and class (if any) it belongs to, the name of the method, a list of parameters (if any), and the statements it carries.

[3]The current version of PYREF also supports the detection of RENAME CLASS, MOVE CLASS, and EXTRACT VARIABLE. However, these refactoring types are not thoroughly tested at this moment. Therefore, in this paper, we focus on method-level refactorings.

[4]https://docs.python.org/3/library/ast.html

[5]https://anytree.readthedocs.io/en/latest/

TABLE I
REFACTORING DETECTION RULES.

Refactoring Type	Rule
Change Method Signature $m1$ to $m2$	$\exists(M, U_1, U_2) = matching(m_1, m_2) \mid m_1 \in m^- \wedge m_2 \in m^+ \wedge m_1.c = m_2.c$ $\wedge((U_1 = \varnothing \wedge U_2 = \varnothing) \vee (\lvert M\rvert >= \lvert U_1\rvert \wedge \lvert M\rvert >= \lvert U_2\rvert \wedge compatibleSignatures(m_1, m_2)) \vee$ $(\lvert M\rvert > \lvert U_1\rvert \wedge \exists inline(m_x, m_2)) \vee (\lvert M\rvert > \lvert U_2\rvert \wedge \exists extract(m_1, m_x))))$ $\boldsymbol{Rename\ Method}$: $m_1.name \neq m_2.name$ $\boldsymbol{Add\ Parameter}$: $\lvert m_2.p\rvert > \lvert m_1.p\rvert$ $\boldsymbol{Remove\ Parameter}$: $\lvert m_2.p\rvert < \lvert m_1.p\rvert$ $\boldsymbol{Change/Rename\ Parameter}$: $\lvert m_2.p\rvert = \lvert m_1.p\rvert \wedge (\lvert set(m_2.p) - set(m_1.p)\rvert > 0 \vee \lvert set(m_1.p) - set(m_2.p)\rvert > 0)$
Extract Method $m2$ from $m1$	$\exists(M, U_1, U_2) = matching(m_1, m_2) \mid (m_1, m_1') \in m^= \wedge m_2 \in m^+ \wedge m_1.c = m_2.c \wedge$ $\neg m_1.calls(m_2) \wedge m_1'.calls(m_2) \wedge \lvert M\rvert >= \lvert U_2\rvert$
Inline Method $m2$ to $m1'$	$\exists(M, U_1, U_2) = matching(m_1, m_2) \mid (m_1, m_1') \in m^= \wedge m_2 \in m^- \wedge m_1'.c = m_2.c \wedge$ $m_1.calls(m_2) \wedge \neg m_1'.calls(m_2) \wedge \lvert M\rvert >= \lvert U_2\rvert$
Move Method $m1$ to $m2$	$\exists(M, U_1, U_2) = matching(m_1, m_2) \mid m_1 \in m^- \wedge m_2 \in m^+ \wedge \lvert M\rvert > \lvert U_1\rvert \wedge \lvert M\rvert > \lvert U_2\rvert \wedge m_1.name = m_2.name \wedge$ $((mod_1, mod_1') \in m^+ \vee (c_1, c_1') \in c^=) \wedge (m_1 \in c_1 \vee m_1 \in mod_1)) \wedge$ $((mod_2, mod_2') \in mod^= \vee (c_2, c_2') \in c^=) \wedge (m_2 \in c_2 \vee m_2 \in mod_2))$ $\boldsymbol{Pull\ Up\ Method}$: $m_1.c.extends(m_2.c)$ $\boldsymbol{Push\ Down\ Method}$: $m_2.c.extends(m_1.c)$

M: matched statements, U_i: unmatched statements, m_i: method, c_i: class, mod_i: module, p: parameters
$codeElement'$: code element after revision, $codeElement^=$: matched element, $codeElement^-$: removed element, $codeElement^+$: added element
$matching(m_1, m_2)$ returns a set of matched statements (M) and two sets of unmatched statements (U_1 and U_2) between the method bodies of m_1 and m_2
$compatibleSignatures(m_1, m_2) \Rightarrow m_1.p \subseteq m_2.p \vee m_2.p \subseteq m_1.p \vee \lvert m_1.p \cap m_2.p\rvert \geq \lvert m_1.p \cup m_2.p\rvert \backslash \lvert m_1.p \cap m_2.p\rvert$
$m_1.calls(m_2)$ returns true if m_1 calls m_2, $m_1.c.extends(m_2.c)$ returns true if the class of m_1 extends the class of m_2
$inline(m_x, m_2)$ returns true if m_x is inlined into m_2, $extract(m_1, m_x)$ returns true if m_x is extracted from m_1

- **Leaf Statement**: A leaf statement is a statement that does not contain nested statements inside. It is associated with all the elements it contains, its parent method (if any), its original AST in text representation, as well as its depth and the index in the AST.
- **Composite Statement**: A composite statement contains the same attributes of a leaf statement. However, a composite statement may contain a body which consists of other statements (*e.g.,* `If` statement).

C. Matching Code Elements

These code elements are categorized into three groups: 1) the common set (*i.e.,* the code element in the current revision can be matched to its counterpart in the parent revision); 2) the added set (*i.e.,* the code element is added in the current revision); and 3) the removed set (*i.e.,* the code element is removed in the current revision). For modules, classes, and methods/functions, we use the following heuristics to decide whether two code elements in two revisions match:

Module : $module_1.name = module_2.name$

Class : $class_1.name = class_2.name$

$\wedge\ class_1.module = class_2.module$

$\wedge\ class_1.host\_method = class_2.host\_method$

The host method is null if the class is not defined in a method.

Method : $method_1.name = method_2.name$

$\wedge\ method_1.module = method_2.module$

$\wedge\ method_1.class = method_2.class$

$\wedge\ method_1.params = method_2.params$

We use the same Statement Matching algorithm applied in REFACTORINGMINER to determine whether two statements match. For leaf statements, we consider them matching if they satisfy one of the following conditions:

1) The statements are identical in textual representation and have the same depth in their AST.
2) The statements are identical in textual representation.
3) The statements are identical in textual representation after replacing compatible sub-expressions.

In case of composite statements, passing one of the three conditions is not enough. They should also have at least one matched pair of statements within their bodies. Due to space limitations, we do not elaborate the Statement Matching algorithm here but we point the reader to the REFACTORING-MINER paper [4] to get deeper insights into how this algorithm works.

When one statement is matched to several statements, the priority order is in line with the order of these three conditions. When several statements fit into the same condition, we sort the matched statements in ascending order based on three different attributes (sort level from high to low): their textual similarity (represented with Levenshtein distance), depth in the AST, and index in the body of their method, then we select the first matched statement.

It is important to note that the matched code elements are not fixed and are subject to change depending on the outcome of refactoring detection. For instance, if we detect that class A is renamed to class B, we will remove them from the added and removed sets and add them to the common set.

TABLE II
REFACTORING DETECTION PERFORMANCE PER REFACTORING TYPE.

Refactoring Type	PyRef					Python-adapted RefactoringMiner				
	TP	FP	FN	Precision (%)	Recall (%)	TP	FP	FN	Precision (%)	Recall (%)
Rename Method	109	22	7	83.21%	93.97%	15	1	101	93.75%	12.93%
Add Parameter	104	5	61	95.41%	63.03%	125	1	40	99.21%	75.76%
Remove Parameter	46	2	5	95.83%	90.20%	21	0	30	100.00%	41.18%
Change/Rename Parameter	74	5	-	93.67%	-	-	-	-	-	-
Extract Method	11	0	9	100.00%	55.00%	12	4	8	75.00%	60.00%
Inline Method	4	1	1	-	-	1	0	4	-	-
Move Method	13	1	8	92.86%	61.90%	16	66	5	19.51%	76.19%
Pull Up Method	5	1	1	-	-	2	4	4	-	-
Push Down Method	1	2	0	-	-	1	0	0	-	-
Overall	293	34	92	89.60%	76.10%	193	76	192	71.74%	50.13%

D. Applying Refactoring Heuristics

PyRef uses predefined rules listed in Table I to detect different types of refactoring. As mentioned in the paper of RefactoringMiner [4], following the order of detection rules is important. For example, when a Rename Method and an Extract Method refactoring are applied sequentially on the same method, if we do not detect the Rename Method refactoring first, we will not be able to match the renamed method with the original method, thus they will not be used to further check if any method is extracted.

E. Sorting Candidates

After applying detection rules, one code element might be associated with several potential refactoring instances. In practice, code elements can undergo only a certain amount of some refactoring types. For example, a method can be renamed only once in a commit, but it can be used to extract several methods. When a method signature is matched to multiple potential signatures, we take into consideration the similarity between signatures. The similarity is calculated based on two values: the sum of the Levenshtein distance between the statement texts in their method bodies, and the number of textual-identical statements. In this way, we can eliminate many invalid refactorings and reduce false positives.

IV. Evaluation

The *goal* of this study is to analyze the accuracy of PyRef for detecting refactorings in Python projects. The *context* consists of 573 refactorings reported by PyRef and Python-adapted RefactoringMiner.

A. Research Question

To evaluate the performance of PyRef, we answer the following research question (RQ):

RQ: What is the accuracy of PyRef and how does it compare to the state-of-the-art tool?

B. Context Selection and Data Collection

To answer this RQ, we evaluate the performance of PyRef and the only existing refactoring detection tool for Python, namely Python-adapted RefactoringMiner.

As Python-adapted RefactoringMiner is still under development, we obtained the list of all the detected refactoring cases by an early version of the tool from its authors. We randomly selected three projects, including dit[6], texar[7], and ffmpeg-python[8]. We applied PyRef on these repositories, and for each project, we removed those reported refactorings committed after the latest commit containing refactoring reported by Python-adapted RefactoringMiner to ensure a fair comparison. As a result, PyRef and Python-adapted RefactoringMiner detected 406 and 269 refactorings, respectively, for the refactoring types PyRef supports. As 102 refactoring cases were reported by both tools, in total we collected 573 refactoring instances.

We used a web app to manually validate the detected refactorings. For each refactoring, the web app presented the information of the refactoring (*i.e.,* refactoring type and description) and the link to the corresponding GitHub commit where code before and after changes can be found. Validators (*i.e.,* authors of this paper) needed to inspect the code changes, determine whether the reported refactoring is correct and leave comments if there is anything worth discussing. As the supported refactorings are well-defined and the validators have good knowledge of refactoring, when a validator was very certain whether the refactoring is true/false positive, we did not require a second validator. When there was any doubt or the validator wanted the case to be double checked, a second or even third validator was involve until agreement was reached. In total, 50 cases required a second/third evaluator.

C. Results

Table II shows the accuracy of PyRef and Python-adapted RefactoringMiner. For each type of refactoring and each tool, we report true positive (TP), false positive (FP), false negative (FN), precision, and recall. For refactoring types containing less than 10 instances, we omit the precision and recall. Given the fact that it is impractical to extract all the refactoring operations applied in the projects, we follow the approach used in the work of RefactoringMiner [4].

[6]https://github.com/dit/dit
[7]https://github.com/asyml/texar
[8]https://github.com/kkroening/ffmpeg-python

More specifically, we consider all the refactorings reported by these two tools as the complete dataset. Therefore, the recall reported here is an upper bound. Moreover, as PYTHON-ADAPTED REFACTORINGMINER uses different refactoring types as PYREF does, and there is no one-to-one or one-to-many relation between CHANGE/RENAME PARAMETER in PYREF to refactorings in PYTHON-ADAPTED REFACTORINGMINER, we only report the result of this refactoring type for PYREF. Meanwhile, the overall result does not contain CHANGE/RENAME PARAMETER, either.

Overall, PYREF achieves a precision of 89.6% and a recall of 76.1%, while PYTHON-ADAPTED REFACTORINGMINER achieves a precision of 71.74% and 50.13%. The performance on different refactoring types also varies. For example, the precision of PYREF on detecting RENAME METHOD (83.21%) is among the lowest, while PYREF has a low recall (55%) for EXTRACT METHOD. On the contrary, PYTHON-ADAPTED REFACTORINGMINER has a low precision (19.51%) for MOVE METHOD and a low recall (12.93%) for RENAME METHOD. Given the fact that there are not many instances detected for some refactoring types (*i.e.,* INLINE METHOD, PULL UP METHOD, PUSH DOWN METHOD), it is hard to tell whether the accuracy for these types can be generalized. This is also due to the fact that these types of refactorings might be less frequently applied. The result that significant fewer refactorings were detected for these types is in line with REFACTORINGMINER. As the overall recall of both tools has some room for improvement, we can find that while both these tools are highly relevant to REFACTORINGMINER, they can still complement each other.

We manually inspected the false positive cases obtained by these tools to get some insights on how they can be improved. For PYREF, many false positives of RENAME METHOD are due to the fact that PYREF does not consider the reserved methods in Python. For example, in the commit 57abf6e[9], PYREF reported that in the class `OutputNode` of the file `ffmpeg/nodes.py`, the method `__init_fromscratch__` (line 306) was renamed to `__init__`. While all the statements in these two methods can be matched and they satisfy the rule of RENAME REFACTORING, it is not a real renaming operation as `__init__` method is reserved in Python functioning as a constructor. Therefore, this operation is in fact moving all the statements to the new method and remove the overridden `__init__` method. The false positive of REMOVE PARAMETER is mainly because we did not consider the magic variable `**kwargs`, which allows users to pass multiple keyword arguments (*e.g.,* `arg1=1, arg2=2`) to a method, when retrieving the parameters. We plan to fix these issues in the next version of PYREF.

For PYTHON-ADAPTED REFACTORINGMINER, we found that most false refactorings reported for MOVE METHOD are due to a bug when converting Python code to Java. For example, in the commit 215319a[10], the tool reported that in the file `txtgen/core/layers.py`, the method `_common_default_conv_kwargs` from the class `PyDummyClass1` is moved to the class `PyDummyClass2`. In fact, these two classes do not exist and the function `_common_default_conv_kwargs` is not moved. As the function is not contained in any class, the tool created a dummy class to feed it into REFACTORINGMINER. For some reason, different classes were created for the same function. The authors of the tool acknowledged that they have noticed this bug, and it has already been fixed in the unreleased new version. Therefore, we believe this tool can obtain a higher precision when the new version is released.

[9]https://bit.ly/3rN7iK2

[10]https://bit.ly/37eCiJu

D. Threats to Validity

Threats to construct validity concern the relation between the theory and the observation. In this work, the threats are mainly due to the measurements we performed. First of all, the oracle is manually labeled, and human errors might exist during validation. To mitigate this threat, we have more than one person to inspect cases in doubt. As most refactoring instances are clear and uncontroversial, we believe this should not be a factor which can hugely impacts the results. Besides, the construction of our oracle is done by incorporating the output of two different tools; we acknowledge that it might miss some real refactorings, thus inflating the recall. Given the fact that refactoring operations are rarely documented, it is extremely difficult to identify all the refactorings applied in a project.

Threats to external validity concern the generalization of our findings. In this evaluation, we only validate the refactoring detection results on three real-world Python projects. We are aware that the precision and recall might vary when PYREF is applied on different projects. In the future, we plan to verify the performance on more projects. Moreover, in this study we focused on method-level refactorings, thus it is unclear whether the heuristics-based approach also works well for refactoring types. However, given the good performance of REFACTORINGMINER, where we get inspiration from, we believe that PYREF is likely to achieve a reasonable accuracy for newly added refactoring types in the future.

Threats to internal validity concern internal factors to our study that could influence our results. The two tools selected in this study are highly relevant to REFACTORINGMINER: PYREF is inspired by REFACTORINGMINER and adopts its core idea for refactoring detection, while PYTHON-ADAPTED REFACTORINGMINER converts Python code to Java and uses REFACTORINGMINER to detect refactoring. That is, these two tools share certain similarities. Tools leveraging other refactoring detection approaches might produce very different results, thus impacting the recall calculated in our study.

E. Replication

To facilitate replication and advancement of refactoring detection for Python, we open source PYREF and release the dataset used in this study, which can be accessed at https://github.com/PyRef/PyRef.

V. Limitations

While PyRef takes a first step to detect refactorings in Python projects, several limitations exist in this tool.

A. Unsupported Refactorings

Currently, PyRef mainly detects method-level refactorings, and much more work needs to be done to extend the support for class-level (*e.g.,* Rename Class) and field-level refactorings (*e.g.,* Pull Up Field). It has been several years since the emergence of the first refactoring detection tool for Java, and the latest version of the state-of-the-art tool RefactoringMiner now supports 80 types of refactorings. However, efforts to support refactoring detection in Python projects has just started, thus needing several years' iteration to become more mature and robust.

One major difference between Java and Python is that the latter is dynamically-typed. That is, by simply parsing the AST, we are not able to know the type of the data. Therefore, PyRef currently does not support type-related refactorings (*e.g.,* Change Parameter Type). One potential solution to this issue is integrating type inference techniques [15], [16].

Besides, there are many refactoring types which are exclusively applied in Python. For example, as Python supports several programming paradigms, investigating how source code is changed from functional programming to objected oriented programming is an interesting topic. Moreover, in Python, some `for` loops can be converted into list comprehensions, which is not possible in other languages like Java. The Python-specific syntax enables new research opportunities.

B. Reliance on Python ast Module

PyRef is written in Python 3 and relies on built-in ast module to parse the source code. That is, when executing PyRef with Python 3 on some old projects written in Python 2, sometimes there will be compatibility issues and parsing errors. While the support of Python 2 has been discontinued since the first day of 2020, adding an extra parser for Python 2 will be helpful to reduce the chance of code parsing issues for legacy code.

C. Refactorings Spanning Multiple Revisions

Like most refactoring detection tools, PyRef detects refactorings between two adjacent revisions. If a refactoring operation spans several revisions, namely the new revision does not contain complete refactoring, PyRef might fail to detect the refactoring. While using non-adjacent revisions for refactoring detection might solve this issue, it will significantly increase the workload of the tool, thus increasing the time needed for generating the results. A better approach is needed to compare only relevant revisions.

VI. Conclusions

We presented PyRef, a tool for detecting refactoring operations in Python projects. PyRef is the first native solution for Python, and it does not require converting the source code to another programming language, nor does it rely on other refactoring detection tools. Our evaluation, which uses real-world projects and compares our approach and the state-of-the-art Python-adapted RefactoringMiner, showed that PyRef achieves high precision. The limitations of PyRef discussed in Section V outlines the roadmap for our future work.

Acknowledgments

We thank Malinda Dilhara and Danny Dig for providing information and refactoring detection results of the early version of Python-adapted RefactoringMiner. We also gratefully acknowledge the financial support of JSPS and SNSF for the project "SENSOR" (No. 183587, JPJSJRP20191502).

References

[1] M. Fowler, *Refactoring: improving the design of existing code (2nd Edition)*. Addison-Wesley Professional, 2018.

[2] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at Xerox," in *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 348–357.

[3] E. 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, 2011.

[4] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[5] D. Silva, J. Silva, G. J. De Souza Santos, R. Terra, and M. T. O. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[6] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A refactoring reconstruction tool based on logic query templates," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, 2010, p. 371–372.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the Object-Oriented Programming*, ser. ECOOP' 06, 2006, pp. 404–428.

[8] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence," *Information*, vol. 11, no. 4, 2020.

[9] M. Dilhara, "Discovering repetitive code changes in ML systems," in *2021 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - Student Research Competition*, ser. ESEC/FSE 2021, 2021, p. 1683–1685.

[10] Z. Xing and E. Stroulia, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08, 2008, p. 951–952.

[11] ——, "UmlDiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, p. 54–65.

[12] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[13] K. De Volder, "Type-oriented logic meta programming," Ph.D. dissertation, Citeseer, 1998.

[14] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*, ser. MSR '17, 2017, pp. 269–279.

[15] J. Aycock, "Aggressive type inference," in *the 8th International Python Conference*, 2000, pp. 11—-20.

[16] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *the 24th ACM SIGSOFT International Symposium on Foundations of Software (FSE)*. ACM, 2016, pp. 607–618.