Finding Functionally Equivalent Methods in Python Using Automated Test Generation Techniques

Yusheng GUO†, Shiyu YANG†, Akihiro TABATA†, and Yoshiki HIGO†

† Graduate School of Information Science and Technology, Osaka University
1–5, Yamadaoka, Suita-shi, 565–0871 Japan

E-mail: †{guoysh,yangsy,a-tabata,higo}@ist.osaka-u.ac.jp

Abstract As a popular programming language in modern software development, Python boasts an extensive open-source codebase on GitHub. Code reuse is common across these vast repositories. This study leverages open-source Python projects from GitHub and applies automated testing techniques to discover functionally equivalent method pairs. The research involved collecting and processing methods from 5.1k Python projects on GitHub. Due to the lack of type checking in Python, grouping methods present specific challenges. To address this, we performed detailed type inference on the methods and grouped them based on the inferred types, providing a structured and comprehensive foundation for further analysis. Automated test generation techniques were applied to create unit tests for each method. These methods were executed against one another within their respective groups to identify candidate method pairs that produced identical outputs given the same inputs. Finally, through manual checking, we identified 130 functionally equivalent method pairs from a pool of 2,400 candidates.

Key words functionally equivalent methods, source code analysis, dataset, code clone

1. Introduction

With the evolution of programming languages, modern languages have become increasingly rich and complex in their syntactical features, particularly dynamic languages like Python. Python, a popular programming language, is widely embraced by developers worldwide due to its concise and readable syntax and powerful standard library. It has a vast codebase and active user communities on platforms like GitHub. Python supports multiple programming paradigms, including object-oriented, functional, and imperative programming. This versatility enables the implementation of identical functionality through different approaches, depending on developers' coding preferences.

The open-source community provides a vast array of software projects containing rich and diverse code resources. It's common to find methods within these codebases that, while functionally equivalent, are implemented in different ways. Collecting such functionally equivalent code snippets is highly valuable for software engineering research. These snippets can be utilized to create datasets of equivalent methods, which can, in turn, drive advancements in areas like code optimization, refactoring, and test generation. However, identifying and collecting functionally equivalent methods remains a complex challenge due to the variations in their structure.

Many existing code clone detection tools rely on identifying repetitions in code snippets to detect clones, such as the token-based SourcererCC [1] and the tree-based DECKARD [2]. These tools are generally effective at identifying syntactically similar code fragments, such as directly copied-and-pasted code or code with minor changes in variable names. However, they often struggle to detect functionally equivalent but structurally different codes. This is because these tools primarily focus on superficial code similarities and overlook functionality. Therefore, there is an urgent need to develop new techniques and tools capable of identifying functionally equivalent code pairs rather than just syntactically similar fragments.

This study's primary goal is to collect functionally equivalent method pairs from open-source projects. Functionally equivalent methods, referred to as FE methods, are defined as pairs of methods that return the same output given the same input (parameters). The key idea of this research is to use Pynguin [3] to automatically generate test cases for the extracted methods, followed by mutual execution to identify methods that exhibit identical behavior under the generated test cases. Subsequently, we manually check all potential FE method pairs to identify the valid FE method pairs. This study selects the ManyTypes4Py [4]dataset as the target for detecting FE method pairs in Python. ManyTypes4Py contains approximately 5.1K type-checked Python repositories, comprising around 1.5 million methods. From this dataset, we extract methods and perform type inference, followed by grouping based on the type inference results. Test cases are then automatically generated, and mutual execution is conducted within each group. Ultimately, we obtained 7415 candidate FE method pairs and manually checked them.

2. Definition of FE Methods

FE methods refer to methods that may differ in implementation but are equivalent in functionality. FE methods are characterized by producing identical outputs when provided with identical inputs. Although their code structures may vary, such as using different algorithms, data structures, or coding styles, they ultimately achieve the same functionality. The concept of functional equivalence is especially important in code optimization, refactoring, and clone detection, as identifying FE methods can help developers understand potential redundant code or opportunities for improvement within a codebase.

Code clone detection is typically categorized into four types based on similarity and structural differences:

- **Type-1 Clones**: Identical code fragments, except for variations in whitespace, comments, or identifier names.
- **Type-2 Clones**: Code fragments that are largely similar but may include changes in identifiers, such as variable names or function names, and some code formatting modifications.

Type-3 Clones: Structurally similar code with some differences in code fragments or logic modifications.

Type-4 Clones: Code fragments that have the same functionality but different implementations, also known as semantic clones. These clones are not based on superficial code similarity but rather on the behavior or functionality of the code.

Traditional code clone detection tools primarily focus on detecting Type-1 and Type-2 Clones, which depend on code structure and syntax similarity. These tools identify clones by searching for similar code fragments, but their limitation lies in their inability to effectively detect code fragments that are functionally identical but structurally different.

This research aims to overcome this limitation by using automatic generation techniques to detect functionally equivalent but differently implemented code clones, specifically Type-4 Clones. Type-4 Clones are particularly challenging, as they do not depend on syntactical similarities but instead require an analysis of method behavior to establish functional equivalence. Detecting these clones is crucial for code refactoring and optimization, as it reveals code fragments that are entirely different in implementation but identical in functionality.

3. Key Idea for Automatically Identifying Candidate FE Method Pairs from FEMPDataset

Previous research in this field has explored techniques such as automatic test case generation and mutual execution methods. The literature [5] proposed an approach to obtaining a set of FE methods by mutually executing the generated test cases. Additionally, a dataset of FE method sets was constructed using Borge's dataset [6]. It contains 276 FE method pairs. Similarly, the FEMPDataset [7], created using similar approaches, consists of 1,342 FE method pairs in Java, validated by three independent programmers.

In the research on FEMDataset, FE method pairs are automatically collected by leveraging both the static features (e.g., method signatures) and dynamic behavior (test results) of Java methods. Static features include return types and parameter types, with methods sharing the same features grouped together. The EvoSuite tool is then used to generate test cases for methods within the same group. Test cases generated by automated test generation techniques have the property that the test cases always succeed. Mutual execution of these test cases is performed to determine whether the methods exhibit equivalent behavior. If a method can pass the test cases generated for another method, and vice versa, the two methods are considered functionally equivalent. Finally, manual checking is conducted to confirm the valid functional equivalence of method pairs, despite differences in implementation. The success of the FEMPDataset highlights the effectiveness of utilizing automatic test case generation tools and mutual execution techniques to identify FE methods. Its success primarily stems from the ability to verify whether two methods produce identical outputs given the same inputs.

Given Python's widespread use and growing importance across various application domains, extending the key ideas of FEMPDataset to Python has significant research and practical value. Python's flexibility and ease of use result in diverse implementations of functions and methods. However, its dynamic nature, which lacks static type checking, presents unique challenges in code clone detection and functional equivalence analysis. Nonetheless, Python offers a wealth of libraries and tools that support automated test case generation and type inference, providing a solid foundation for detecting FE method pairs. Therefore, this research proposes to extend these ideas to Python.

4. Procedure of Dataset Construction

In this study, the following process is used to construct a dataset of FE method pairs:

STEP-1: Extract Python methods from open-source projects on GitHub and perform an initial filtering.

STEP-2: Perform type inference on each method and group them accordingly.

STEP-3: Generate test cases for each method.

STEP-4: Mutually execute methods within the same group to identify candidate FE method pairs.

STEP-5: Manually check each candidate FE method pair to confirm if they are valid FE method pairs.

Figure 1 provides an overview of the five steps described above. STEP-1 through STEP-4 is automatically performed by the developed tool, while only Step 5 is executed manually. The detailed process for each step is as follows:

4.1 STEP-1

To build the dataset of FE method pairs, we selected the ManyTypes4Py(1) dataset as the basis for our experiment. This is a Python benchmark dataset for machine learning-based type inference, containing 5,382 Python projects sourced from GitHub. It offers a diverse collection of open-source Python projects, covering various types and application scenarios and providing an abundant sample of methods. A key reason for choosing this dataset is its suitability for type inference.

From the selected Python projects, we extracted all Python methods. Initially, we used Python's Abstract Syntax Tree (AST) module to parse all `.py` files within the projects. This allowed us to construct the abstract syntax tree for each file and extract method definitions. We then traversed these abstract syntax trees to gather method-related information. In total, we obtained 1,500,000 Python methods. For each method, the following information was collected and recorded in a database: method name, (original) source code, normalized source code, the number of statements and conditional predicates, file path, start line, and end line.

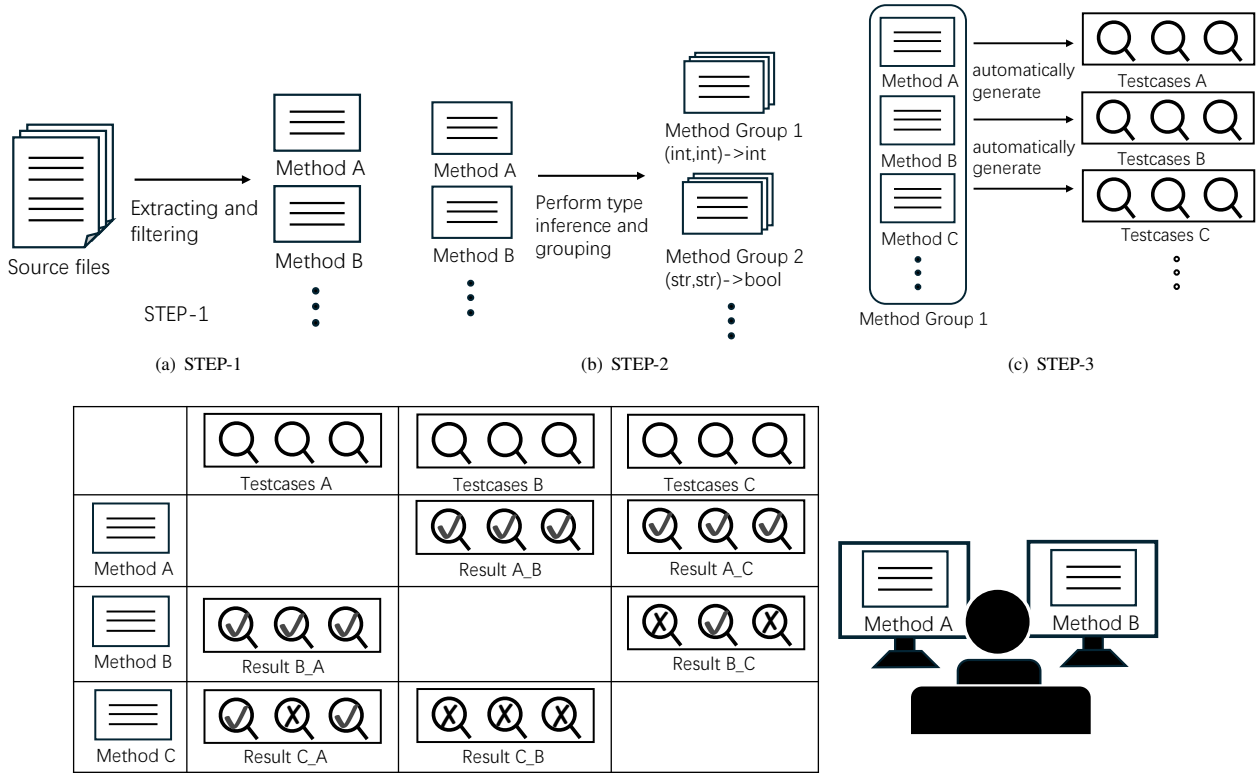
After processing the extracted Python methods, we normalized the source code. This process involved standardizing all variables, string constants, and code indentation using the `ast` library. These steps helped eliminate formatting differences and excluded the impact of different variable names. Due to the extensive standard library of built-in types in Python, we chose not to normalize the names of method calls.

Figure 2 shows an example of normalization. Subfigure 2(a) shows the original method code, and subfigure 2(b) shows the code after normalization. During the normalization process, we first removed the type hints and default values from the method declarations. Next, we renamed all variables, attribute names, and string literals. For all method calls, only the method itself and its recursive calls were renamed.

The reason for not renaming other method calls is that Python has many powerful built-in methods, and renaming all method calls might result in different methods being incorrectly identified as duplicate code. Additionally, any code that calls user-defined external methods is excluded in subsequent steps, as it is beyond the scope of this study.

After completing the code normalization, we generated a unique hash value for each normalized method code. By calculating the hash value of the method code, we could effectively detect and identify duplicate methods. The primary purpose of this step is to eliminate any potential duplicate methods in the dataset, ensuring

(1): It was presented in the data showcase of the MSR'21 conference.



(a) STEP-1 (b) STEP-2 (c) STEP-3

(d) STEP-4 (e) STEP-5

Fig. 1 Steps to obtain pairs of functionally equivalent Python methods

```
def get_open_business_day(business, day):
    "\n    Helper function which returns 'day' dictionary of
corresponding day for\n given business dictionary. If the day
is not found, returns None.\n    "
    if (len(business.open_hours) == 0):
        return None
    for open_day in business.open_hours:
        if (open_day.day == day):
            return open_day
    return None
```

(a) Original Method

```
def func(val1, val2):
    if len(val1.attr1) == 0:
        return None
    for val3 in val1.attr1:
        if val3.attr2 == val2:
            return val3
    return None
```

(b) Normalized Method

Fig. 2 Example of normalization

```
def crossOff(possible, prime):
    nextPrime = None
    for i in range(prime, len(possible)):
        if possible[i] % prime == 0:
            possible[i] = 0
        if possible[i] and (not nextPrime):
            nextPrime = possible[i]
    return nextPrime
```

(a) Original Method

```
def crossOff(possible: list, prime: int) -> int:
    nextPrime = None
    for i in range(prime, len(possible)):
        if possible[i] % prime == 0:
            possible[i] = 0
        if possible[i] and (not nextPrime):
            nextPrime = possible[i]
    return nextPrime
```

(b) Method after type inference

Fig. 3 Example of type inference

that each method pair in the dataset is unique.

Next, we filtered the remaining methods to remove those not meeting our research requirements. This step ensured that only relevant methods were retained in the dataset. The following types of methods were excluded:

- **Methods with no parameters or return values**: These methods cannot be effectively evaluated through test cases, as their behavior cannot be adequately judged. Consequently, they did not provide helpful information for functional equivalence analysis and were excluded.
- **Methods with `self` in their parameters**: Methods containing the `self` parameter are typically instance methods in object-oriented programming. Since our research focuses on generating unit tests for standalone methods, instance methods were removed from the dataset.
- **Methods that invoke external classes or methods**: To ensure that the methods in our dataset are self-contained and can be independently analyzed, any method that calls external classes or methods was filtered out. These methods would fail during automatic test case generation, so they were removed in advance to better facilitate the detection of FE method pairs.

By applying these filtering criteria, we ensured that the remaining methods in the dataset are better suited for further analysis and the identification of candidate FE method pairs. After completing these steps, 28,353 methods were retained for subsequent analysis.

4.2 STEP-2

Since Python is a dynamically typed language and lacks static

```
@pytest.mark.xfail(strict=True)
def test_case_2():
    int_0 = -1741
    int_1 = 2067
    int_2 = module_0.inv(int_1, int_0)
    assert int_2 == -1740
```

(a) Test case with the 'xfail' marker

```
def test_case_0():
    bool_0 = True
    set_0 = {bool_0, bool_0, bool_0}
    module_0.set_add(set_0, set_0)
```

(b) Test case without assert statements

Fig. 4 Example of removed test cases

type checking, this poses challenges for subsequent operations. To improve the effectiveness of automated analysis, it is necessary to perform type inference on methods and group them based on the inferred types. Methods with explicit type information tend to perform more reliably in automated test case generation, making type inference essential.

In this study, we used the TypeT5 [8] tool for type inference. TypeT5 is a Transformer-based model specifically designed for type inference in Python code. It can infer the types of variables and parameters and return values directly from the source code, particularly excelling when explicit type hints are absent. Leveraging TypeT5 allows for a better understanding of method type information, providing a solid foundation for subsequent test generation and functional equivalence detection.

In the subsequent automated test case generation, we will rely on type hints to generate test cases. If the type hints for a method include Python non-built-in types, the test case generation will fail outright. Therefore, before performing inference, developer-provided type hints were initially processed, with non-built-in type hints being removed. Next, we used the TypeT5 tool to infer the types of parameters and return values for methods lacking explicit type hints. After completing the type inference, We rechecked each method's parameters and return values to ensure they all belong to Python's built-in types. Methods that still contain non-built-in types in parameter types or return value types will be removed. This step ensured that the final retained methods had clear built-in type information. Ultimately, 21,503 methods were preserved for grouping and the next step of automated test case generation.

Figure 3 presents an example of type inference. In the original code, the methods lack any type hints. After applying type inference, as shown in the red-highlighted section of the figure, we annotate the variable possible as a list, prime as an int, and the return value as an int. Based on these results, we grouped the methods accordingly. This method is grouped together with other methods that have parameter types (list, int) and a return type of int.

Following this, we grouped all methods based on their parameters and return types. Only methods with identical parameters and return types were grouped together. In the end, there were 726 groups, and each group contained at least two methods. In STEP-4, we will perform mutual execution within these groups.

4.3 STEP-3

In this step, our primary goal was to generate corresponding test cases for all the methods. We chose the Pynguin [3] for automatic test case generation. Pynguin is a Python-based automatic test case generation tool that can produce a comprehensive set of test cases for a given method, ensuring that all functional aspects of the method are thoroughly tested. By utilizing Pynguin, we could automatically generate test cases for all methods obtained in the previous steps, ensuring each method was adequately validated.

After generating the test cases, we performed initial filtering to remove those test cases that contained 'xfail' markers or lack of assert statements. In Figure 4, we present a test case marked with 'xfail' and another test case lacking an assert statement. The 'xfail' marker indicates that the test case is expected to fail, which generally means that it cannot effectively validate the correctness of the method, making it unsuitable for functional equivalence analysis. On the other hand, test cases lacking 'assert' statements cannot verify whether the method's actual output matches the expected results. They only checked whether the method could run correctly under given inputs, and therefore, these cases were also excluded. This filtering process aims to ensure that the test cases effectively validate the behavior of the methods rather than merely executing code without actual verification.

Next, we conducted coverage testing on the remaining test cases using the coverage component provided by pytest. The purpose of coverage testing is to assess the extent to which the test cases cover the method's code. To ensure the effectiveness and comprehensiveness of the test cases, we retained only those with 100% coverage. This means that these test cases can cover all code branches and paths in the method, ensuring no code segments that could impact functionality are omitted. Test cases with 100% coverage provide the most thorough validation, ensuring that the functional equivalence analysis in subsequent steps is based on complete and accurate test results.

Ultimately, after this filtering and coverage testing step, we obtained a high-quality set of test cases, including a total of 6,500 test cases for methods. This step took approximately 40 hours. These test cases can comprehensively and accurately validate the functional behavior of each method. We will use these methods and their corresponding test cases for mutual execution in the next step.

4.4 STEP-4

In this step, we utilize the high-quality test cases and method information retained from Step-3, as well as the grouping information from Step-2, to perform mutual execution among methods within the same group. The primary objective of this step is to validate whether the methods are functionally equivalent through cross-testing.

The detailed process is as follows:

(1) **Preparation of Test Cases and Methods**:
Suppose we have method A and method B, each with corresponding test cases A and test cases B. These test cases were rigorously filtered in Step-3 to ensure that they have 100% coverage, thereby thoroughly assessing the functionality of the methods.

(2) **Execution of Tests**:
First, we execute method A using test cases B. This step aims to evaluate the performance of method A under the test cases from method B, confirming whether method A can pass all the tests in test cases B. If method A passes all the tests in test cases B, we proceed to the next step: executing method B using test cases A. This step is intended to verify the performance of method B under the test cases from method A.

Through this approach, we conduct cross-testing between methods A and method B to ensure that both methods can pass under different test cases.

(3) **Result Analysis**:
If method A passes all tests in test cases B and method B passes all tests in test cases A, it indicates that methods A and B are likely functionally equivalent under the given test cases. This result suggests that both methods produce identical outputs for the same inputs and may be functionally equivalent. Therefore, we mark this pair of methods as candidate FE method pairs.

Conversely, if any test case fails during the testing process, it indicates a functional discrepancy between methods A and B under the given test cases. In this case, we exclude this method

pair from the candidate FE method pairs to ensure that only method pairs that consistently perform similarly across all test cases are considered functionally equivalent.

For method pairs marked as candidate FE method pairs, further validation and analysis are required. Although the preliminary mutual execution provides initial evidence of functional equivalence, the test cases are limited, and this only indicates functional equivalence under specific conditions. In practical applications, additional verification steps are necessary to ensure that these method pairs exhibit consistent behavior across all possible input conditions.

4.5 STEP-5

In this phase, we perform a detailed manual checking of all candidate FE method pairs. The primary objective of this step is to confirm whether these method pairs indeed exhibit functional equivalence through human judgment.

After completing the mutual execution step, we obtained a list of candidate FE method pairs. Each pair was selected based on the cross-testing results, where both methods passed all tests in each other's test cases. Although this result provides preliminary evidence of functional equivalence, further validation is required to confirm this equivalence.

For candidate method pairs identified as functionally non-equivalent during manual checking, we created new test cases to highlight their functional differences. We designed test inputs that could potentially cause the two methods to produce different results and executed these newly created test cases on both methods. If the methods produce different outputs for the same test case, it indicates that their behavior diverges under certain conditions, thereby demonstrating a functional difference between them.

Finally, we record those method pairs that are confirmed to be valid FE method pairs and use these pairs to construct a dataset.

5. Dataset

In this section, we describe the dataset we constructed. The dataset is built using source code from ManyTypes4Py [3], which includes approximately 5.1k open-source Python projects. From these, we extracted a total of 1,500,000 methods to build our dataset. In STEP-1, based on research requirements, we retained 28,356 methods for type inference. In STEP-2, these methods were grouped into 726 categories according to the results of the type inference. After filtering out groups with only one method, we proceeded with the subsequent steps. In STEP-3, we used Pynguin to generate test cases for the remaining methods automatically. After processing the test cases and removing those with 'xfail' markers and those without assert statements, we rechecked that the coverage of the test cases was 100%. Ultimately, we had 2,434 methods and their corresponding test cases that met the requirements for mutual execution. These were divided into 129 groups, with the largest group containing 528 methods. In STEP-4, we identified a total of 7,415 potential FE method pairs. These pairs were then subjected to manual checking. We manually checked 750 candidate FE method pairs and identified 150 valid FE method pairs.

The number of candidate FE method pairs obtained in STEP-4 is quite large. Due to the limitations of automatically generated test cases, it is challenging to detect functional differences in methods that involve string manipulations. This also applies to methods that return boolean values, as it is difficult to capture edge cases with a limited number of test cases. This introduces some challenges for manual checking.

To address this, we adopted the following approach to extract a subset of candidate FE method pairs for manual checking: For each method pair, if neither method has been inspected in any previous pair, the pair is selected for checking. The current pair is skipped if either method has already been included in a previously inspected pair. After this filtering process, the number of method pairs requiring checking was reduced to 750. We spent approximately 10 hours manually checking these pairs, ultimately identifying 130 valid FE method pairs.

```
def sum1d(s:int, e:int) -> int:
    c = 0
    for i in range(s, e):
        c += i
    return c
```

(a) Method sum1d

```
def while_count(s:int, e:int) -> int:
    i = s
    c = 0
    while i < e:
        c += i
        i += 1
    return c
```

(b) Method while_count

Fig. 5 Example of FE method pairs

In the end, We constructed the dataset and published it on GitHub. The dataset consists of three tables: `methods`, `pairs`, and `verifiedpairs`. The methods table records all relevant information about each method, including the original method, its normalized version, the number of lines in the method, the test cases generated for the method, and the method's grouping information (see Table 1). The pairs table contains all the candidate equivalent method pairs obtained in STEP-4. Each method pair is linked to the corresponding original methods in the methods table based on the pair's information. Additionally, every candidate method pair has a unique ID. The verifiedpairs table records the IDs of the method pairs that have been manually verified as functionally equivalent.

Figure 5 shows an example of FE method pairs identified in STEP-5. Both methods calculate the sum of all integers from `s` to `e-1` but differ in their implementation.

The `sum1d` method uses a `for` loop to iterate over all integers from `s` to `e-1`, with range `(s, e)` generating a sequence of integers from `s` to `e-1`. The for loop automatically iterates through this sequence.

In contrast, the `while_count` method uses a while loop for accumulation. The while loop requires manual updating of the loop

Table 1 Schema for the `methods` Table

Column Name	Data Type	Description
signature	STRING	Method signature
name	STRING	Method name
rtext	BLOB	Raw text of the method
ntext	BLOB	Normalized text of the method
size	INT	Lines of code
branches	INT	Number of branches
hash	BLOB	Hash of ntext
path	STRING	File path of the method
start	INT	Start line of the method
end	INT	End line of the method
repo	STRING	Project repository name
revision	STRING	not used in this dataset
compilable	INT	The methods used in STEP-4
tests	INT	not used in this dataset
Target_ESTest	BLOB	Automatically generated test cases
Target_Tesecase	BLOB	The test cases used in STEP-4
groupID	INT	Group identifier for the method
id	INTEGER	Method ID

```
def gcd(a: int, b: int) -> int:
    while a != 0:
        (a, b) = (b % a, a)
    return b
```

(a) Method gcd

```
def mutated_gcd(a: int, b: int) -> int:
    if a < b:
(a, b) = (b, a)
    while b != 0:
        (a, b) = (b, a % b)
    return a
```

(b) Method mutated_gcd

Fig. 6 Example of functionally non-equivalent method pairs

variable `i` and checking the loop condition `i < e`. Here, `i` starts from `s`, and `i` is incremented by 1 in each iteration.

Figure 6 shows an example of functionally non-equivalent method pairs identified in STEP-5. Both methods are designed to calculate the greatest common divisor (GCD) of two integers using the same algorithm—the Euclidean algorithm. However, differences in specific implementation details lead to divergent outputs for some inputs.

The `gcd` method is a classical implementation of the Euclidean algorithm, which iteratively swaps `(a, b)` in a while loop under the condition `a` is not equal to 0 until `a` becomes 0, at which point it returns `b`. The `mutated_gcd` method also implements the Euclidean algorithm but adds an if statement at the start to ensure that `a` is always greater than or equal to `b`. If `a` is less than `b`, the values of `a` and `b` are swapped. The swapping continues inside the while loop until `b` equals 0, at which point it returns `a`.

When both input parameters are either positive or negative, the two methods return the same result regardless of the values of `a` and `b`. However, when `a` and `b` have opposite signs and `a` is greater than `b`, the results differ in terms of their signs. This discrepancy arises because the termination condition of the while loop in the two methods depends on different variables `gcd` relies on `a`. In contrast, `mutated_gcd` relies on `b`. For instance, with the input `(12, -8)`, the `gcd` method returns `4`, while the `mutated_gcd` method returns `-4`. This difference went undetected in the test cases primarily because the `mutated_gcd` method only checks the relative magnitude of `a` and `b` and does not account for the signs of the numbers.

6. Related work

This research is inspired by the FEMPdataset study [7]. In FEMPdataset's work, a dataset of 1,342 FE method pairs in Java was constructed by automatically generating test cases and mutual execution. This paper primarily extends that approach to Python, with several key differences outlined below.

- The IJADataset used in FEMPdataset contains approximately 314 million lines of code, from which 23 million methods were extracted. In contrast, this study uses the ManyTypes4Py database, extracting 1.5 million methods. Additionally, the size of the constructed datasets differs: FEMPdataset includes 1,342 FE method pairs, whereas the dataset in this paper contains 130 FE method pairs.
- In FEMPdataset, Java method types were directly used for grouping. However, since Python lacks static type checking, this study uses TypeT5 for type inference to facilitate the grouping and mutual execution process.
- In FEMPdataset, test execution was skipped when fewer than five test cases were generated. In this study, no such limitation was imposed. Due to differences in the test case generation tools, this study checked test case coverage, retaining only those test cases with 100% branch coverage.
- During the final manual checking phase, FEMPdataset's candidate functionally equivalent pairs were evaluated independently by three individuals. In this study, I conducted the visual checking alone. However, for pairs deemed non-equivalent, I generated new test cases to demonstrate their functional differences.

7. Conclusion

In this study, we extracted Python methods from open-source projects and automatically generated test cases for them. These generated test cases were then mutually executed to identify candidate functionally equivalent method pairs. We manually checked a subset of these candidate FE method pairs. Ultimately, from 7,400 candidate functionally equivalent pairs, a total of 750 pairs were selected for manual verification, of which 130 pairs were confirmed to be functionally equivalent. In the future, we plan to utilize this dataset for additional research in code clone detection, such as evaluating existing clone detection tools.

Currently, one of the primary challenges of this research is the large number of candidate functionally equivalent method pairs. Manual checking of all these pairs is impractical. This issue primarily arises from the low quality of the automatically generated test cases. To address this, we plan to develop a better filtering process for the test cases to reduce the number of method pairs requiring manual checking. Enhancing the quality of the test cases will be crucial in improving the efficiency and accuracy of identifying functionally equivalent method pairs.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP24H00692, JP21K18302, JP21H04877, JP23K24823, and JP22K11985.

References

[1] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp.1157–1168, 2016.

[2] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," 29th International Conference on Software Engineering (ICSE'07), pp.96–105, 2007.

[3] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," 44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE, pp.168–172, ACM/IEEE, May 2022.

[4] A.M. Mir, E. Latoskinas, and G. Gousios, "Manytypes4py: A benchmark python dataset for machine learning-based type inference," IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp.585–589, IEEE Computer Society, May 2021.

[5] Y. Higo, S. Matsumoto, S. Kusumoto, and K. Yasuda, "Constructing dataset of functionally equivalent java methods using automated test generation techniques," 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), pp.682–686, 2022.

[6] H. Borges, A. Hora, and M.T. Valente, "Understanding the factors that impact the popularity of github repositories," 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.334–344, 2016.

[7] Y. HIGO, "Dataset of functionally equivalent java methods and its application to evaluating clone detection tools," IEICE Transactions on Information and Systems, vol.E107.D, no.6, pp.751–760, 2024.

[8] J. Wei, G. Durrett, and I. Dillig, "Typet5: Seq2seq type inference using static analysis," The Eleventh International Conference on Learning Representations, 2023.