Published version: Kota Someya, Lei Chen, Michael J. Decker, Shinpei Hayashi, "How Much Can a Behavior-Preserving Changeset Be Decomposed into Refactoring Operations?,", in Proceedings of the 41st IEEE International Conference on Software Maintenance and Evolution, pp. 809–814, 2025, https://doi.org/10.1109/ICSME64153.2025.00084

How Much Can a Behavior-Preserving Changeset Be Decomposed into Refactoring Operations?

Kota Someya
School of Computing
Institute of Science Tokyo
Tokyo, Japan
someya@se.comp.isct.ac.jp

Lei Chen
School of Computing
Institute of Science Tokyo
Tokyo, Japan
chenlei@se.comp.isct.ac.jp

Michael J. Decker
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
mdecke@bgsu.edu

Shinpei Hayashi
School of Computing
Institute of Science Tokyo
Tokyo, Japan
hayashi@comp.isct.ac.jp

***Abstract*—Developers sometimes mix behavior-preserving modifications, such as refactorings, with behavior-altering modifications, such as feature additions. Several approaches have been proposed to support understanding such modifications by separating them into those two parts. Such refactoring-aware approaches are expected to be particularly effective when the behavior-preserving parts can be decomposed into a sequence of more primitive behavior-preserving operations, such as refactorings, but this has not been explored. In this paper, as an initial validation, we quantify how much of the behavior-preserving modifications can be decomposed into refactoring operations using a dataset of functionally-equivalent method pairs. As a result, when using an existing refactoring detector, only 33.9% of the changes could be identified as refactoring operations. In contrast, when including 67 newly defined functionally-equivalent operations, the coverage increased by over 128%. Further investigation into the remaining unexplained differences was conducted, suggesting improvement opportunities.**

***Index Terms*—refactoring, behavior preservation**

I. Introduction

Refactoring is a common practice in software development to improve code readability/maintainability. It involves modifying the internal structure of code while preserving its external behavior [1]. However, in real-world software development practices, developers may mix behavior-preserving modifications (e.g., refactoring) with behavior-altering modifications that introduce functional changes, i.e., *floss refactoring* [2], [3]. It may result in commits with *tangled changes*, which hinder the review process [4], [5].

Addressing this, a series of authors propose refactoring-aware code review approaches [6]–[8]. These approaches facilitate a clearer understanding of changes during the review process by detecting fine-grained behavior-preserving operations using a refactoring detection tool so that the behavior-preserving modifications can be separated from behavior-altering modifications. In essence, such refactoring-aware approaches are expected to be particularly effective when behavior-preserving parts can be extracted from a difference by using a sequence of behavior-preserving operations, including those detectable by refactoring detection tools, thus allowing developers to focus on the remaining behavior-altering parts. However, in practice, the concept of behavioral equivalence includes a wide range of operations, from simple renamings to substantial internal structural changes such as algorithm substitution. Therefore, we argue that there are inherent limitations in fully explaining functionally equivalent parts using sequences of behavior-preserving operations.

As such, in this paper, we investigate the usefulness of a refactoring-aware code review approach. Specifically, we focus on functionally equivalent code diffs and quantify how much of them can be decomposed into sequences of generalizable behavior-preserving operations, such as refactoring, and what remains unexplained. In our evaluation, we do not address large-scale refactorings that affect the entire project. Instead, we focus on the limitations of difference decomposition at the method level, using a refactoring detection tool and sequences of behavior-preserving operations.

For the dataset, we utilize the Functionally Equivalent Method Pairs (FEMP) dataset [9], which contains pairs of Java methods collected from open-source software (OSS) that are functionally equivalent. To evaluate the decomposition capability and detection limitations of existing refactoring tools, we use RefactoringMiner [10], [11] as a representative. Results show that a refactoring detector (i.e., RefactoringMinor) can only be used to explain up to 34% of the changes. To provide a more comprehensive explanation of behavior-preserving changes, we introduce a catalog of 67 additional behavior-preserving operations and show that it can explain 77.5% of changes, an increase of 128%. Finally, we discuss the limitations of decomposing functionally equivalent code into behavior-preserving operations.

More formally, we study the following research questions (RQs):

- ***RQ*$_1$**: To what extent can a refactoring detection tool decompose functionally equivalent changes using its existing techniques and catalogs?
- ***RQ*$_2$**: To what extent can a refactoring detection tool decompose functionally equivalent changes when using an iterative approach?
- ***RQ*$_3$**: To what extent can functionally equivalent changes be decomposed via a more comprehensive set of behavior-preserving operations?

Answering these RQs results in the following contributions:

- We demonstrate the decomposition capability of a refactoring detection tool applied as is and iteratively.

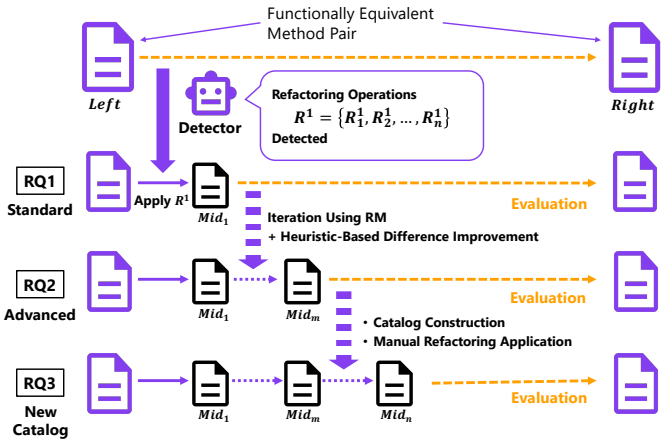


Fig. 1. Overview of the study.

- We introduce an extended catalog of method-level behavior-preserving operations.
- We show that the extended catalog can be used to decompose code changes much more effectively into behavior-preserving operations.

The remainder of this paper is structured as follows. In Section II, we explain our methodology. Section III explains the results of our empirical study. Finally, Section IV provides the conclusion and future work.

II. Methodology

Figure 1 gives an overview of the methodology behind our investigation. Our investigation involves three rounds of study. For round one (i.e., RQ_1), we utilize as input a pair of functionally equivalent methods *Left* and *Right*. We apply a refactoring detector and perform the detected operations on *Left*, resulting in the intermediate code Mid_1. We then evaluate the remaining differences between Mid_1 and *Right*. For the second round (i.e., RQ_2), we rename the methods and parameters of Mid_1 to match those of *Right*, and then iteratively perform refactoring detection and application of those refactorings. This represents a best-case scenario for current refactoring detection approaches. We once again analyze the remaining changes, and we propose an extended catalog of behavior-preserving operations, which we apply in round three (i.e., RQ_3). Finally, we investigate any remaining unexplained differences and provide improvement opportunities.

A. Dataset

In this paper, we use the FEMP dataset [9] to evaluate a refactoring detection tool and construct a more comprehensive catalog. The FEMP dataset consists of pairs of functionally equivalent Java methods collected from a large number of open-source projects. Here, *functionally equivalent* means returning identical outputs for any given input, i.e., exhibiting the same external behavior. Even if the internal algorithms or logic differ significantly, they are still considered functionally equivalent as long as their input-output behavior is identical. We randomly selected 100 pairs from the FEMP dataset.

TABLE I
Refactorings Detectable by RMiner

• Add Method Annotation	• Remove Method Annotation
• Add Variable Modifier	• Remove Variable Modifier
• Change Variable Type	• Rename Method
• Extract Variable	• Rename Parameter
• Inline Variable	• Rename Variable

B. Refactoring Detection Tool

We use RefactoringMiner ver. 3.0.7 (RMiner) [11] as a representative refactoring detector to conduct a detailed analysis of the capabilities of current refactoring detectors in explaining behavior-preserving changes. We selected RMiner because it has shown the highest accuracy among refactoring detection tools in previous studies [12]. In addition, RMiner supports 102 refactoring operations, including minor ones, making it considerably more comprehensive than other tools. As such, cases where RMiner fails to detect certain refactoring operations are likely to suggest issues commonly shared by existing detection techniques.

Only twelve method-level refactorings supported by RMiner were detectable in our dataset. However, we exclude two of them: Add/Remove Parameter, as applying them individually violates behavior preservation, making them false positives. The ten refactorings used are listed in Table I.

C. Refactoring Applier

To evaluate the decomposition capability of detected refactoring operations, we implemented a refactoring applier that automatically applies sequences of such operations. The applier takes the output of RMiner as input and applies them using the Eclipse JDT [13]. Table I also shows the refactoring coverage of the applier. Of the 10 refactorings detected at the method level, 8 out of 10 refactorings can be automatically applied. For Extract Variable and Inline Variable, we decided to apply them manually because the output of RMiner does not include sufficient information to precisely perform these refactorings.

Note that in all cases, we verified that functional equivalence is preserved when applying each operation, regardless of whether it is applied automatically or manually, by running the test cases provided in the FEMP dataset.

D. Evaluation Metric

The following formula is used to quantify how well a set of behavior-preserving operations explains the differences between two functionally equivalent code fragments *Left* and *Right*, with *Mid* representing the application of those operations to *Left*:

$$Sim(Mid, Left, Right) = 1 - \frac{|\Delta(Mid, Right)|}{|\Delta(Left, Right)|}.$$

Here, $|\Delta(c, c')|$ is the size of the delta between code fragments c and c', calculated as the total number of added and deleted tokens obtained using srcDiff [14]. We adopt srcDiff, as it

completely preserves all source code text, making it well-suited for quantifying the size of unexplained differences. Conceptually, *Sim* computes the similarity based on the size of the delta between Mid_k and *Right*, normalized by the original delta between *Left* and *Right*, to assess how much of the delta has been explained. *Sim* ranges from 1, meaning 100% of the changes between *Left* and *Right* are explained by the behavior-preserving operations, to 0, meaning none of the changes can be explained.

III. Empirical Study

The investigation results to answer the three RQs are summarized below. All results, including the application process and intermediate results of operations, are available in the supplemental package [15].

A. RQ_1: Standard Detector Usage

1) Motivation: In this RQ, we investigate the decomposition capability of the refactoring detector in standard usage.

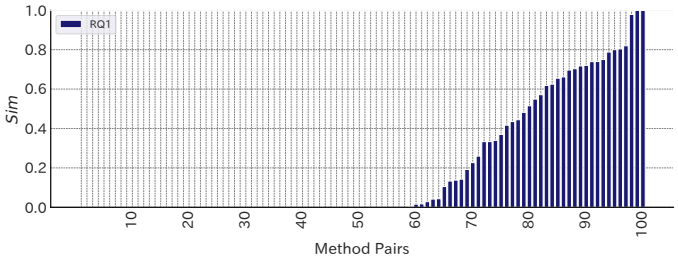
2) Study Design: First, we input a target method pair $(Left, Right)$ into RMiner and obtain a set of detected refactoring operations $R^1 = \{R_1^1, R_2^1, \dots\}$. Next, we sequentially apply the applicable operations from R^1 to *Left* to obtain an intermediate code fragment Mid_1. Finally, we use *Sim* to measure how well Mid_1 explains the differences between *Left* and *Right*.

3) Results: The results are shown in Fig. 2a. This figure presents the *Sim* values for each method pair, sorted in ascending order. 42 of 100 pairs had at least one refactoring operation detected by RMiner that could be applied. The average *Sim* across all method pairs is 0.2. Only two pairs achieved a *Sim* of 1, indicating that all modifications were decomposed via a single round of detection and application.

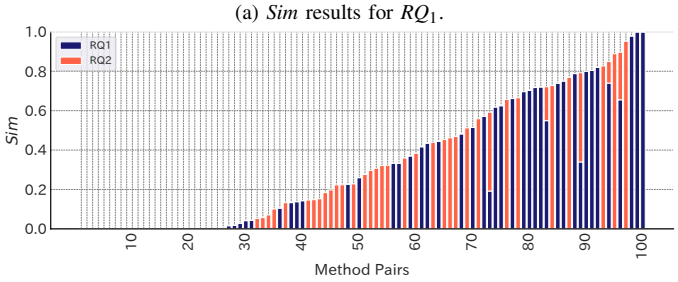
B. RQ_2: Iterative Detector Usage

1) Motivation: Figure 3 shows the detection results by RMiner for the method pair #10998 in the FEMP dataset. In this example, only Rename Method, Inline Variable, Rename Variable, and Rename Parameter are correctly detected. The changes to the first two parameters are incorrectly detected as a combination of Add Parameter and Remove Parameter. Moreover, we identified that issues caused by renaming are not limited to parameter renames. When parameter names and/or method names are different, especially when the internal structures of the code fragments vary significantly, the detector may fail to recognize the pair as corresponding methods, resulting in no detected operations. Thus, unifying the names can enable detection that would otherwise fail.

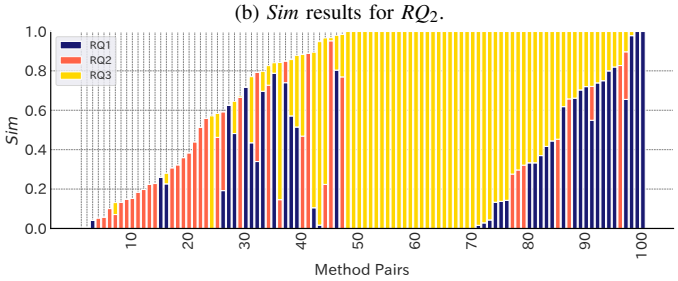
Furthermore, applying the detected operations to obtain a new version and then performing detection again can sometimes yield additional refactorings. For example, in the case mentioned above, reapplying RMiner after applying the initially detected operations results in the correct detection of a previously missed Inline Variable. This result suggests that due to the characteristics of the detector's implementation,



(a) *Sim* results for RQ_1.



(b) *Sim* results for RQ_2.



(c) *Sim* results for RQ_3.

Fig. 2. Changes in *Sim* across all RQs.

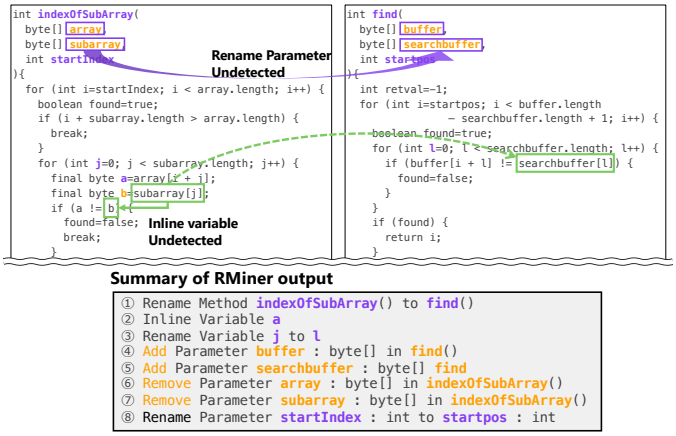


Fig. 3. Insufficient detection by RMiner (#10998).

some refactoring operations may be missed or misinterpreted when the input difference is large. Reducing the size of the difference may improve the detector's performance. In this RQ, we investigate the decomposition capability of the existing detection tool when manually applying rename refactorings and then iteratively using RMiner.

2) Study Design: First, as discussed previously, because a detector may identify inapplicable refactorings (e.g., Add/Remove Parameter) or fail to detect any refactorings

TABLE II
CATALOG OF BEHAVIOR-PRESERVING OPERATIONS

- Apply Constant Folding
- Apply De Morgan's Law
- Apply Negation as Inequality
- Conditional to Expression [16]
- Consolidate Variable Declaration and Initialization
- Decompose Conditional Branch
- Factor Out Coefficient
- Factor Out Inequality as Negation
- Inline Return Variable
- Introduce Cast
- Introduce Constant to Comparison Expression
- Introduce Dead Code
- Introduce Parentheses
- Introduce Return Variable
- Merge Conditional Branch
- Merge Variable Declaration
- Move Statement Across Try
- Remove Branch by Pre-Assignment
- Remove Cast
- Remove Dead Code [1]
- Remove Double Negation
- Remove Parentheses
- Remove Unused Variable
- Replace Array Declaration Style
- Replace Assignment with Compound Assignment
- Replace Compound Assignment with Assignment
- Replace Conditional Operator with If
- Replace For with Foreach
- Replace Foreach with For
- Replace Guard Clause with Conditional
- Replace If with Switch
- Replace Inclusive Comparison with Exclusive
- Replace Nested Conditional with Guard Clauses [1]
- Replace Numeric Representation
- Replace Postfix Increment/Decrement with Prefix
- Replace Prefix Increment/Decrement with Postfix
- Replace Switch with If
- Reverse Comparison Operator
- Reverse Conditional [17]
- Split Chained Assignment
- Split Conditional Branch
- Split Variable Declaration
- Split Variable Declaration and Initialization
- Swap Commutative Operands
- Swap Conditional Branches
- Transpose Equation
- Unwrap Statement from Block [16]
- Wrap Statement in Block [16]

when the method/parameters are renamed, we manually detect and apply Rename Method/Parameter to unify the names. The result of renaming is the intermediate code Mid_2.

After renaming to create Mid_2, we iteratively apply RMiner. First, we apply RMiner to Mid_2 to detect the sequence of operations between $(Mid_2, Right)$. The detected operations are then applied to Mid_2 to obtain Mid_3. This process is repeated until no refactoring operations are detected or no further operations can be applied. Specifically, for a given intermediate code fragment Mid_t, we use RMiner to detect the operations $R^t = \{R^t_1, R^t_2, \dots\}$ between $(Mid_t, Right)$ and sequentially apply R^t to Mid_t to obtain Mid_{t+1}. This process is repeated until R^t becomes an empty set or no further operations can be applied to Mid_t. The final code fragment obtained in this iterative process is denoted as Mid_m and used as the target for evaluation.

3) Results: The results are shown in the same format as RQ_1 in Fig. 2b. 76 of 100 method pairs had at least one refactoring operation that could be applied through the iterative detection process by RMiner. As with RQ_1, only two method pairs achieved a *Sim* of 1, indicating that all differences were fully explained. The average *Sim* across the 100 pairs is 0.339, representing a marked increase from 20.0% for RQ_1.

C. RQ_3: Expansion of Behavior-Preserving Operations

1) Motivation: Figure 4 shows the detection results by RMiner for the method pair #11311 from the FEMP dataset. In this example, an `else` block is removed and replaced with a guard-clause. However, this operation is not defined in RMiner and thus remains undetected. As demonstrated in RQ_1 and RQ_2, one reason for the limited decomposition capability is the insufficient variety of refactoring operations. To address this, we investigate the extent to which decomposition capability can be improved by introducing a new catalog with additional behavior-preserving operations. We also examine the extent to which differences remain unexplained even after applying sequences of these operations.

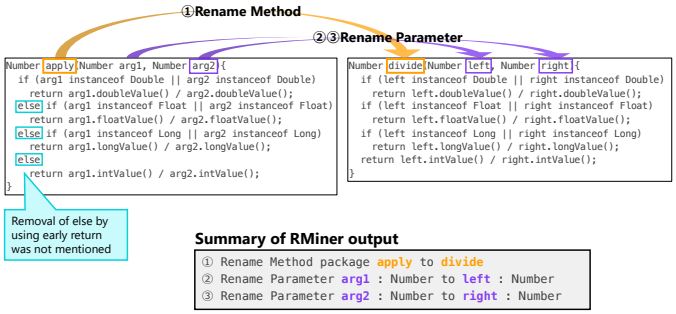


Fig. 4. Example of an operation not implemented in RMiner (#11311).

2) Study Design: One of the authors created a catalog with behavior-preserving operations by observing the remaining differences between the intermediate code fragment Mid_m and *Right* (after applying the procedures for RQ_1 and RQ_2). Based on these observations, we define additional behavior-preserving change operations. In constructing the catalog, we consider factors such as the frequency of occurrence, the feasibility of normalization, and the length of the code after normalization. Operations that are considered general-purpose based on the authors' judgment are selected.

For the code fragment Mid_m obtained from RQ_2, we compare it to *Right* and manually detect the additional behavior-preserving operations. We iteratively apply these operations until no further operations from the extended catalog can be applied. This results in the final code fragment Mid_k. Then, we calculate the *Sim* for Mid_k, *Left*, and *Right*.

3) Results: In the experiment conducted for RQ_2, 66.1% of the differences remained unexplained. One of the authors carefully examined these remaining differences and constructed a catalog of 67 additional behavior-preserving operations that are not defined in RMiner. Each operation in the catalog is structurally normalized based on transformations conforming to the Java language specification [18]. Figure 5 shows the

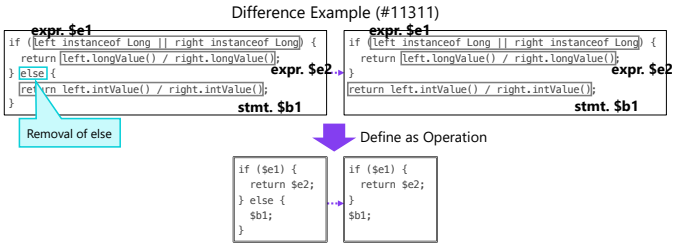


Fig. 5. Extracting Replace Nested Conditional with Guard Clauses.

process of identifying the operation of Replace Nested Conditional with Guard Clauses from an actual example and normalizing it for the catalog. Note that the same code fragment referenced (e.g., `$e1`) in the code before and after the modification is not required to be identical. Rather, it is expected that the code fragments are functionally equivalent.

48 of the operations included in the catalog are listed in Table II. These are operations not defined in RMiner, some of which may overlap with existing catalogs; we did our best to add references for those that have already been introduced in existing catalogs. The remaining 19 operations not shown in the table are related to library usage, such as replacements between `StringBuilder` and `StringBuffer`. These can be found in the complete catalog [15].

It should be noted that the operations defined here are behavior-preserving, but do not necessarily improve the readability/maintainability, e.g., Introduce Dead Code. Therefore, we deliberately avoid calling them "refactoring operations" and instead refer to them as "behavior-preserving operations." Also, these operations were directly extracted from our investigation of the dataset we examined and include well-known, representative changes. We do not claim to have invented these changes, nor do we present this list as exhaustive.

The results are shown in Fig. 2c, in the same format as in previous RQs. Throughout the procedure, at least one operation was detected by RMiner or through manual detection in all 100 method pairs. Furthermore, 53 of the 100 method pairs achieved a final *Sim* of 1, indicating that all modification differences were fully decomposed through the sequence of behavior-preserving operations. The average *Sim* for the 100 method pairs is 0.775.

> The average *Sim* in RQ_1, RQ_2, and RQ_3 was 0.200, 0.339, and 0.775, respectively. These results indicate that introducing a catalog of behavior-preserving operations leads to an increase of 57.5% over the single use of a refactoring detection tool, and an increase of 43.6% over the iterative application of a detection tool. This suggests that existing approaches to supporting change understanding can be further improved by refining detection tool usage and expanding the catalog. However, 22.5% of the changes remained unexplained, indicating room for improvement.

D. Threats to Validity

a) Internal Validity: The definition, manual detection, and application of operations in RQ_3 involve subjective judgments solely by the first author, which may lead to a risk of misidentifying behavior-altering changes as behavior-preserving and/or overlooking operations that should have been detected. In both manual and automated applications, although we used the test cases provided by the dataset to ensure behavior preservation, there may still be untested boundary conditions that go undetected, resulting in incorrectly concluding functional equivalence.

b) External Validity: Changes extracted from the FEMP dataset may be different from real-world projects. Changes often occur at the class or project level, and a variety of programming languages may be involved. As such, the generalizability of the results to other contexts remains uncertain.

IV. Conclusion

In this paper, we investigated the extent to which differences between functionally equivalent method pairs from the FEMP dataset can be decomposed using sequences of behavior-preserving operations. Specifically, we examined the decomposition capability of the refactoring detection tool RefactoringMiner under three conditions: standard usage (RQ_1), iterative usage (RQ_2), and with an expanded catalog of behavior-preserving operations (RQ_3). When applied to 100 method pairs, standard usage resulted in an average *Sim* of 0.200. With the iterative application of detection and application, the *Sim* increased to 0.339. Furthermore, by incorporating an extended catalog of additional behavior-preserving operations, the *Sim* reached 0.775. Despite these improvements, 22.5% of the differences remained that could not be fully decomposed through sequences of behavior-preserving operations.

Based on the obtained results, we discuss the following future directions:

Further expansion of the catalog. The current catalog remains incomplete. By analyzing a broader range of method pairs, we aim to expand its coverage and diversity. We also plan to include behavior-preserving operations beyond the method-level, such as those at the class and project levels.

Automated detection and application. In this study, the detection and application of newly defined operations were conducted manually. To enable broader empirical studies and provide practical support, such as refactoring-aware code review, it is necessary to automate the identification and application of such operations from code differences.

New approaches to change decomposition. To promote understanding where the decomposition of differences proved challenging, it is important to explore alternative approaches instead of the decomposition of primitive operations.

Acknowledgments

This work was supported in part by JSPS Grants-in-Aid for Scientific Research (JP23K24823, JP25K03102, JP25H01125, JP24H00692, and JP21KK0179) and US National Science Foundation: CNS 22-32593/32594.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Addison-Wesley, 2018.

[2] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008.

[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, 2012.

[4] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013)*, 2013, pp. 121–130.

[5] F. Coelho, T. Massoni, and E. L.G. Alves, "Refactoring-aware code review: A systematic mapping study," in *Proceedings of the 3rd IEEE/ACM International Workshop on Refactoring (IWoR 2019)*, 2019, pp. 63–66.

[6] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill, "Refactoring-aware code review," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017)*, 2017, pp. 71–79.

[7] S. Hayashi, S. Thangthumachit, and M. Saeki, "REdiffs: Refactoring-aware difference viewer for Java," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*, 2013, pp. 487–488.

[8] R. Brito and M. T. Valente, "RAID: Tool support for refactoring-aware code reviews," in *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC 2021)*, 2021, pp. 265–275.

[9] 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.

[10] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE 2018)*, 2018, pp. 483–494.

[11] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[12] L. Tan and C. Bockisch, "A survey of refactoring detection tools," in *Proceedings of the Workshops of the Software Engineering Conference, CEUR-WS*, vol. 2308, 2019, pp. 100–105.

[13] Eclipse Foundation, "Eclipse Java development tools (JDT)," https://projects.eclipse.org/projects/eclipse.jdt.

[14] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, "srcDiff: A syntactic differencing approach to improve the understandability of deltas," *Journal of Software: Evolution and Process*, vol. 32, no. 4, pp. e2226:1–31, 2020.

[15] K. Someya, L. Chen, M. J. Decker, and S. Hayashi, "Artifact for "How much can a behavior-preserving changeset be decomposed into refactoring operations?"," Zenodo, 2025. [Online]. Available: https://doi.org/10.5281/zenodo.16443373

[16] L. Chen, M. Lanza, and S. Hayashi, "Understanding code change with micro-changes," in *Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME 2024)*, 2024, pp. 363–374.

[17] B. Murphy and M. Fowler, "Reverse Conditional," https://www.refactoring.com/catalog/reverseConditional.html, 2006.

[18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java language specification, Java SE 11 edition," https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf, 2018.