Published version: Lei Chen, Michele Lanza, Shinpei Hayashi, "Understanding Code Change with Micro-Changes,", in Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution, pp. 363–374, 2024, https://doi.org/10.1109/ICSME58944.2024.00041

Understanding Code Change with Micro-Changes

Lei Chen
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
chenlei@se.c.titech.ac.jp

Michele Lanza
Software Institute
Università della Svizzera italiana
Lugano, Switzerland
michele.lanza@usi.ch

Shinpei Hayashi
School of Computing
Tokyo Institute of Technology
Tokyo, Japan
hayashi@c.titech.ac.jp

***Abstract*—A crucial activity in software maintenance and evolution is the comprehension of the changes performed by developers, when they submit a pull request and/or perform a commit on the repository. Typically, code changes are represented in the form of code diffs, textual representations highlighting the differences between two file versions, depicting the added, removed, and changed lines. This simplistic representation must be interpreted by developers, and mentally lifted to a higher abstraction level, that more closely resembles natural language descriptions, and eases the creation of a mental model of the changes. However, the textual diff-based representation is cumbersome, and the lifting requires considerable domain knowledge and programming skills. We present an approach, based on the concept of *micro-change*, to overcome these difficulties, translating code diffs into a series of pre-defined change operations, which can be described in natural language. We present a catalog of micro-changes, together with an automated micro-change detector. To evaluate our approach, we performed an empirical study on a large set of open-source repositories, focusing on a subset of our micro-change catalog, namely those related to changes affecting the conditional logic. We found that our detector is capable of explaining more than 67% of the changes taking place in the systems under study.**

***Index Terms*—software evolution, source code mining, diff analysis**

I. Introduction

Understanding the code changes made by developers during the submission of pull requests and/or commits to a repository is a critical activity in the maintenance and evolution of software [1]. It can help developers track the state of the software system [2], maintain the current code [3], pinpoint when and where an issue is introduced [4], and plan for future development.

The code changes are usually represented as textual code differences (diffs), which provide a detailed account of which lines and characters of source code have been added, removed, or modified between two file versions. An example of code diff found in a commit in the open source repository *HikariCP*[1] is shown in Fig. 1. The side-by-side code comparison illustrates modifications, where the left side code represents the pre-update version and the right side displays the post-update version, with red highlighting deletions and green indicating additions. The diff view does little to help in understanding that the nearly 40 lines involved in this change were the result of the developer inverting a conditional expression, and subsequently moving a piece of code: there are too many trees depicted here to see the forest.

The diff view, while precise at a textual level, does not meet the cognitive needs of developers when understanding code changes [5], as developers do not think in terms of characters and lines, but at a higher abstraction level, forming a mental model of the logic behind the code [6]. Developers need to compare the code on the left and right sides line by line and interpret the diffs by mentally lifting them to a higher abstraction level, closer to a natural language narrative that encapsulates the essence of the change operations [7]. The lifting process is time and energy-consuming [8] and also requires extensive domain knowledge and proficient programming skills [9]. Though commit messages are designed to reveal the code changes contained in commits, they often are of low quality and cannot convey the change well [10].

To overcome the above difficulty and mitigate the gap between the raw, textual-level diffs and the deeper, conceptual understanding developers need, we propose the concept of *micro-changes*. Micro-changes are a set of code change operations described in natural language, designed to bridge the cognitive divide by translating the textual diffs into more understandable natural-language described operations. As opposed to code diffs, which operate at a low semantic level, focusing exclusively on the literal character alterations, micro-changes lift the semantic level of code diff by distilling their essence into comprehensible, natural-language descriptions. The contributions of this paper are:

- The definition of micro-changes to model and explain code changes.
- A catalog of 20 types of conditional-related micro-changes.
- An automated micro-change detector.
- An empirical study on 73 open source Java repositories, finding that 67.1% of the conditional-related code changes can be covered by the detected micro-changes.

II. Related Work

A. Modern Code Review

Code review is a key practice for ensuring software quality and facilitating maintenance, serving as checkpoint for identifying issues and fostering code improvements [11], [12]. Modern code review is a lightweight, tool-based way focusing

[1]https://github.com/brettwooldridge/HikariCP/commit/2260cc2

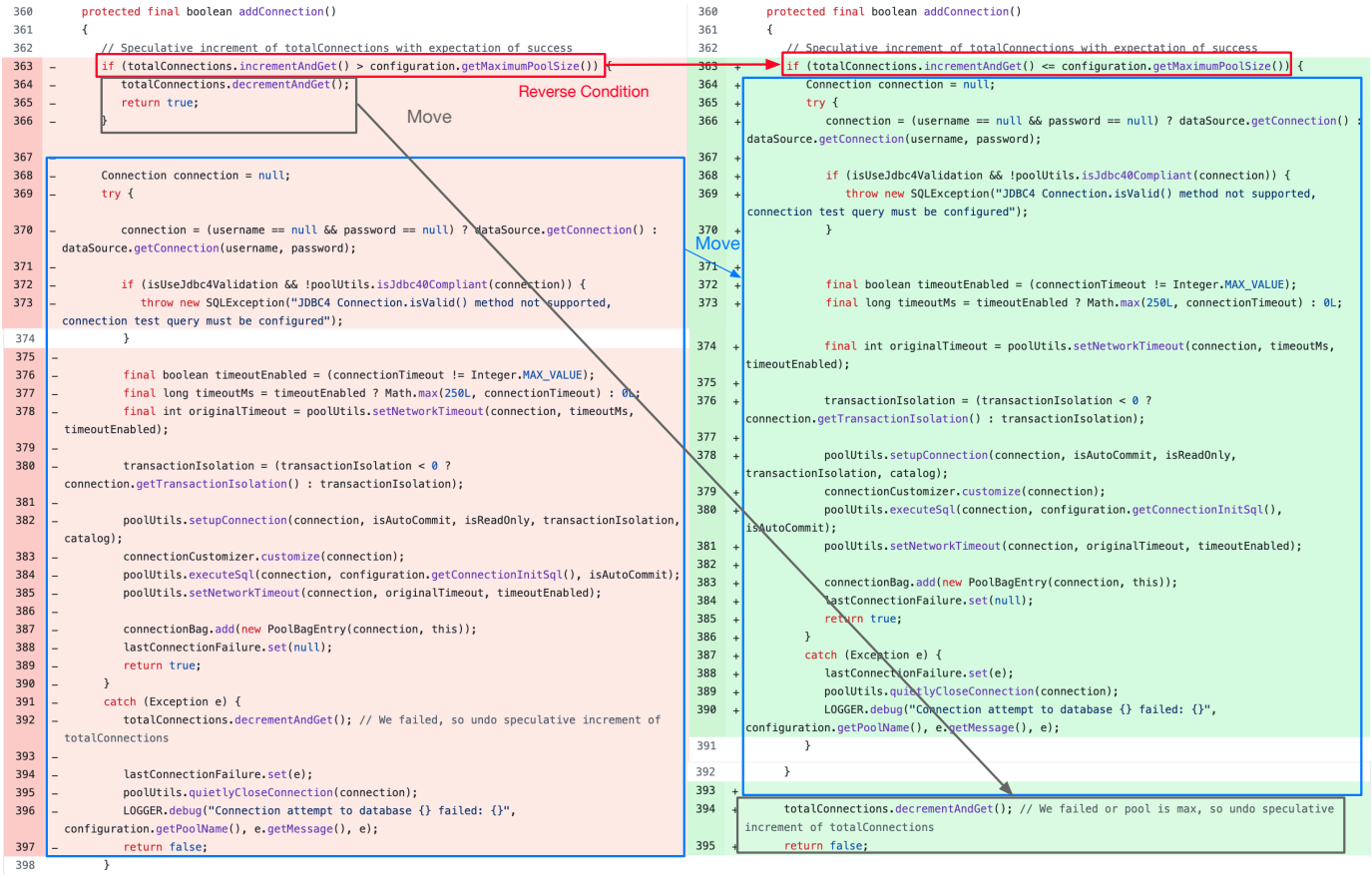

Fig. 1. Example of the code diff in a commit in repository *HikariCP*.

on code changes and is widely adopted for both industrial [1], [2], [13] and open source software [14].

This process typically starts with a developer submitting a pull request that involves single or multiple *commits*, which is a set of changes, deletions or additions to the code encapsulated in a single update. These changes are presented as a *diff* (difference), showing the alterations between the new and existing code, highlighting the additions, deletions, and modifications.

Other team members review the diff, providing feedback, suggestions, and approval, ensuring the code change is optimal in terms of quality, functionality, and adherence to project standards before it is merged into the main branch.

The diff being reviewed can be categorized into two types: *text-based diff* and *tree-based diff*.

B. Text-Based Diff

The text-based diff is widely used in computing differences between two versions of a source file [15], [16]. It shows the added, deleted, and changed text lines.

Canfora et al. [17], [18] introduced a line differencing methodology, termed *ldiff*, that possesses the capability to monitor the positions of lines irrespective of the programming languages involved. This methodology initially employs the Unix diff algorithm to determine the lines that remain unchanged. Subsequently, it utilizes a combination of set-based and sequence-based metrics to facilitate the comprehensive mapping of the remaining lines. Asaduzzaman et al. [19] proposed *LHDiff*, which adopts the *simhash* [20] technique to speed up the mapping process and employs a set of heuristics to improve the effectiveness of tracking source locations.

However, the text-based diff approach fails to effectively leverage the inherent structure of source code.

C. Tree-Based Diff

The tree-based diff utilizes the abstract syntax tree (AST) of the source code and tree differencing algorithms to extract more detailed change information.

Fluri et al. [21] proposed a tree difference algorithm, which can find the matching nodes between two ASTs and calculate the minimum edit script that transforms one AST to another.

Faller et al. [22] proposed a tool named *GumTree* that also takes the "move" action into account in addition to the insertion, deletion, and update actions. They evaluated their tool on a large-scale dataset to measure its accuracy.

However, the tree-based diff, while precise and detailed, requires knowledge about ASTs to understand. Additionally, it is at a low abstraction level which is not intuitive and does not necessarily align with how developers conceptualize code changes.

TABLE I
CONDITIONAL-RELATED MICRO-CHANGE CATALOG OVERVIEW

Micro-change	Description
Add Conditional Statement	Add a new conditional statement.
Add Conjunct or Disjunct	Modify an existing conditional statement by appending an additional condition using logical AND (`&&`) or logical OR (`||`) operators to refine or expand the criteria for the statement's execution.
Adjust Condition Boundary	Modify a comparison operator in a conditional statement to alter its boundary condition.
Conditional to Boolean Return	Simplify a conditional statement that directly returns a boolean value by replacing the if statement with a direct return of the condition's evaluation.
Conditional to Expression	Simplify an if-else statement into a concise conditional operator.
Conditional to Switch	Transform a series of if statements into a switch statement.
Extend Else with If	Replace a simple else clause in an if-else statement with an else if condition, adding a new conditional check to the existing control flow for more specific action selection based on multiple conditions.
Extend If with Else	Add an else clause to an existing if statement.
Flip Logic Operator	Alter the logical operator within a conditional statement, switching between AND (&&) and OR (||).
Move inward Condition	Reorganize nested conditional statements by moving one of the conditions from an outer if statement to be combined with the condition of an inner if statement.
Move outward Condition	Reorganize nested conditional statements by moving one of the conditions from an inner if statement to be combined with the condition of an outer if statement.
Remove Conditional Statement	Remove an existing conditional statement.
Remove Conjunct or Disjunct	Modify an existing conditional statement by removing conditions from a compound logical expression that concatenates multiple conditions by logical AND (`&&`) or logical OR (`||`).
Remove Else	Remove the else clause from an if-else statement, leaving only the if statement and its associated action.
Reverse Condition	Invert the logic of a conditional expression within an if statement, changing the condition to its logical opposite.
Swap Then and Else	Swap the actions of the then and else clauses of an if-else statement.
Unwrap Statement from Block	Remove the curly braces from a block containing a single statement following an if statement.
Unwrap Statement from Conditional	Remove the conditional check around a statement, so the statement executes unconditionally, independent of the previously specified condition.
Wrap Statement in Block	Enclose a single statement within curly braces following an if statement.
Wrap Statement in Conditional	Wrap an existing statement, within an if statement.

D. Refactoring

Refactoring is the concept at a higher level of abstraction by emphasizing systematic modifications that enhance the internal structure of the code while preserving its external behavior [23], [24], and it is a key practice in agile development processes [25].

To detect refactorings, Silva et al. proposed RefDiff [26], [27], which detects refactorings through two phases: source code analysis and relationship analysis. Initially, it parses the source code to construct a model of high-level entities, e.g., types, methods, and fields. Then, it assesses relationships within this model across code changes to identify refactorings by matching these relationships against predefined rules.

Tsantails et al. introduced RefactoringMiner [28], [29], marking the first refactoring mining tool that operates without the need for any code similarity thresholds. This tool detects refactorings through AST-based statement matching with predefined rules. The code entities in the two revisions are matched in top-down order based on text similarity.

This concept aligns with the need for a more abstract representation of code changes. However, the inherent limitation of refactoring, its focus on behavior-preserving transformations, restricts its applicability, leaving a wide range of common developer tasks without a suitable abstraction.

E. Refactoring-Aware Code Review

By recognizing refactorings during the code review process, parts of the textual difference can be consolidated into coherent refactoring operations, thereby increasing the review's efficiency [30]. Hayashi et al. [31], [32] proposed an interactive code difference viewer that can separate refactoring changes from other changes. Ge et al. [33] developed a tool that excludes refactoring-related code changes by reading the refactoring history and displaying the remaining code changes. RefDistiller [34] is a tool checking the existence of missing edits or extra edits when developers manually conduct refactorings to detect potential behavioral changes. Brito et al. [35] propose RAID, which is a refactoring-aware code review tool based on the refactoring detector RefDiff [26]. They conducted a field study with eight professional developers, and they found that RAID can reduce the cognitive effort required for reviewing refactorings when using textual diffs. Due to the inherent constraints of refactoring, many code changes are beyond its descriptive reach. Adopting a micro-change-aware approach to code review can further enhance efficiency.

III. MICRO-CHANGES

A. Definition

Based on the need for an intermediary semantic layer within the code modification hierarchy, we present the concept of *micro-change*.

Micro-changes are code change operations described in natural language, designed to bridge the cognitive divide by translating the textual diffs into more understandable natural-language described operations.

It offers a more efficient and understandable way of examining code changes, encompassing not only refactorings but also a broader spectrum of modifications. The usage of

micro-changes aims to redefine how developers interpret and communicate code changes, facilitating a deeper understanding and more efficient conveyance of their essential nature.

An example micro-change (*Reverse Condition*) is shown in Fig. 1; the core change consists only of one line of code (line 363) that inverses the condition of an if statement, while the diff view misleads one to think that there are dozens of changed lines.

Establishing a complete and exhaustive catalog of micro-changes is the wider context of our work, and goes beyond the scope of this paper. Here we focus on a specific category of micro-changes, namely those related to changes pertaining to conditional expressions, such as if statements. Those changes are selected because they involve fundamental and essential aspects of logic control in software, which significantly impacts the understanding of behavior through a change. Micro-changes are also applicable to different types of changes other than conditional expressions.

B. Catalog of Conditional-Related Micro-Changes

We have developed a catalog comprising 20 distinct micro-change types, each designed to represent common code change patterns affecting conditional expressions of if statements. Due to space constraints within this paper, we provide a concise overview of the catalog in Table I, which contains only the name and the description of our catalog. The complete version is published for reference[2].

We also provide an example of a micro-change *Add Conjunct or Disjunct* in the catalog shown in Table II. It contains

1) *Name*: a concisely described name in natural language,
2) *Description*: a description that outlines the nature of the micro-change,
3) *Structure*: the structure, explaining the micro-change in pseudo-code,
4) *Motivation*: a motivation for the existence of this micro-change,
5) *Example*: one or more examples taken from open-source repositories, to illustrate the micro-change in practice, and
6) *Detection Rule*: a specific detection rule designed to identify this micro-change in the wild.

The construction of the catalog went hand-in-hand with the detection of micro-changes, following a multi-step iterative process, which we explain in Section IV.

C. Research Questions

To have a deeper understanding of the proposed micro-changes and the catalog built, we propose two research questions as follows.

***RQ*$_1$:** *To what extent can the designed micro-changes explain the code change?* The purpose of RQ_1 emerges from the necessity to assess the comprehensiveness and completeness of the developed conditional-related micro-change catalog. It can also uncover opportunities for future enhancements.

[2]https://github.com/salab/Micro-Change-Catalog

TABLE II
CATALOG ENTRY FOR *Add Conjunct or Disjunct*

Add Conjunct or Disjunct

Description
Modify an existing conditional statement by appending an additional condition using logical AND (`&&`) or logical OR (`||`) operators to refine or expand the criteria for the statement's execution.

Structure

- `if` ($\$c_1$) → `if` ($\$c_1$ `&&` $\$c_2$)
- `if` ($\$c_1$) → `if` ($\$c_1$ `||` $\$c_2$)

where `if` $(\cdots)$ represents an if statement, c_1 and c_2 denotes two conditional expressions. The `&&` and `||` denote the logical AND and logical OR operators, respectively.

Motivation
Attach extra conditions to refine or expand the criteria for the statement's execution with logical AND (`&&`) or logical OR (`||`) operators, respectively.

Example
An example of *Add Conjunct or Disjunct* found in the repository *zuul*[3]:

```
107 -         if (server != null) {
108               this.host = server.getHost();
109               this.port = server.getPort();
110               this.availabilityZone = server.getZone();
```

before change

```
107 +         if (server != null && server != DiscoveryResult.EMPTY) {
108               this.host = server.getHost();
109               this.port = server.getPort();
110               this.availabilityZone = server.getZone();
```

after change

The new condition `server != DiscoveryResult.EMPTY` is attached to the `server != null` to make the condition more strict by requiring an extra criterion to be met.

Detection Rule
Detection rules are introduced in Section IV.

***RQ*$_2$:** *What are the frequencies of the micro-changes types?* By quantifying how frequently each micro-change type occurs, RQ_2 investigates their prevalence and distribution across repositories. Furthermore, identifying the frequency of various micro-changes can help prioritize which types to focus on for further refinement in the catalog and enhancement of detection algorithms.

IV. DETECTING MICRO-CHANGES

To validate the catalog of micro-changes, ensure its completeness, and further investigate the features of micro-changes, we designed a process for detecting the micro-changes in the catalog in the wild.

A. Detection Process

The overview of the micro-change detection process is shown in Fig. 2. Starting from the initial input, a Git repository is transformed into a method-level repository, where pairs of pre-change and post-change methods are extracted. These pairs undergo analysis by a tree-diff tool to generate *edit scripts* and *actions*. Concurrently, refactorings are extracted from the Git repository, within which the *Rename*-related refactorings

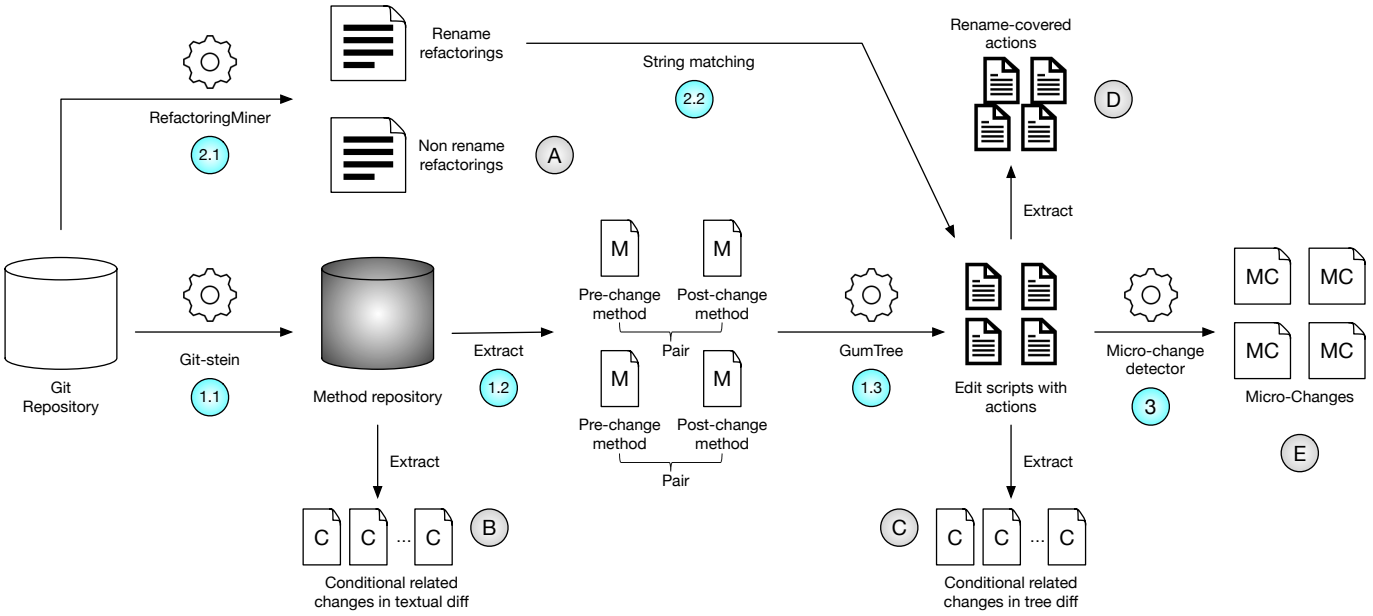

Fig. 2. Overview of micro-change detection.

are used to identify the actions that can be annotated as renaming. By excluding these renaming actions, the micro-change detector detects micro-changes in the remaining. The subsequent sections detail each step of this process:

1.1: Derive Method-Level Repository*:* The Git repository is converted to a method-level repository, where each method in the original repository is extracted into an individual file. This enables Git to track changes at method level [36], [37], which is the level of our interest since conditional expressions are exclusively found in methods. For the conversion of the repository, we use the tool *git-stein*[4] [38].

1.2: Extract Changed Methods*:* From the method repository we obtained from the last step, a complete history of each method can be extracted. Within this history, methods before and after each commit are identified as method pairs representing the changes. We extract all such pairs from the method history to serve as input for the subsequent analysis step, excluding instances where methods are either newly introduced or entirely deleted. Note that method renaming is treated as a special case of deletion and introduction; the original method is considered deleted, and a new method with the new name is introduced, thereby excluding these renaming events from our analysis of method changes.

1.3: Extract AST Edit Scripts*:* We use GumTree [22] to analyze the pairs of pre-change and post-change methods and extract the edit scripts. The GumTree supports only several programming languages. The idea of micro-change is language-agnostic and can be further extended to be applied to other programming languages by substituting the edit script extraction tool and corresponding predefined detection rules.

[4]https://github.com/sh5i/git-stein

An edit script consists of a series of actions, such as *insert*, *delete*, *update*, and *move*, detailing how to transform the original AST into the modified version.

The *insert* action identifies a new AST node that has been added to the AST, represented as INS(*node*, *parentNode*), where the *node* specifies the newly added AST node and the *parentNode* refers to the node under which the new node is inserted. The *delete* action detects an AST node present in the original AST that is absent in the modified version of AST and denoted as DEL(*node*), where *node* is removed. The *update* action represents a node in the AST that has been altered to the same type but a different value, such as modifying a variable name or changing the literal value. It is denoted as UPD(*node*, *val*), where *node* is the node being altered and *val* is the new value assigned to *node*. The *move* action signifies that an AST node be relocated within the AST, maintaining its original structure but changing its location in the code. It is denoted as MOV(*node*, *parentNode*), where *node* is the moved node, *parentNode* is the node above the moved node at the post-change AST.

We exclude the edit scripts consisting of only insertions or only deletions, as they would be part of pure addition or pure deletion commits, and not change-related commits.

2.1: Extract Refactorings*:* Starting from the original Git repository, we use RefactoringMiner 3.0.4 [29] to extract all refactorings performed during its history. However, while it is capable of identifying where refactorings have been applied, it is currently not capable of revealing which other locations in the source code are impacted by refactorings, which is necessary in the specific case of rename refactorings.

To illustrate this point, let us consider the example shown

in Fig. 3. It is a commit from the repository *mbassador*[5], where a *Rename Attribute* refactoring is detected; *delivered* is renamed to *dispatched*, transitioning from line 26 to line 27 post-change. This renaming influences the conditional expressions previously at line 54 and modified to line 57. Despite these changes being a direct result of the refactoring, RefactoringMiner does not identify such downstream effects.

```
26 -     private volatile boolean delivered = false;
54 -         if (!delivered) {
55               if (!isFilteredMessage() && !isDeadMessage()) {
56                   runtime.getProvider().publish(new FilteredMessage(message));
```

before change

```
27 +     private volatile boolean dispatched = false;
57 +         if (!dispatched) {
58               if (!isFilteredMessage() && !isDeadMessage()) {
59                   runtime.getProvider().publish(new FilteredMessage(message));
```

after change

Fig. 3. *Rename Attribute* refactoring.

To address the identified issue, we enhanced the functionality of RefactoringMiner, to also output the code locations affected by a refactoring; if an *update* action showing that value Y is assigned to a code element whose original value is X, and a *Rename*-related refactoring is detected within the same file at the same commit that renames X to Y, we classify this action as a downstream effect of the refactoring.

2.2: Apply String Matching: In commit where the *Rename*-related refactorings are detected, we conduct a string-matching; if the action in the edit script within a commit is updating the name of a code element, and it belongs to the downstream effect of the *Rename*-related refactoring, we regard this action as covered by the *Rename* refactoring. Such actions are excluded from the input considered by the micro-change detector because refactoring constitutes a specific category of micro-changes.

3: Detect Micro-Changes: To detect micro-changes of conditional-related changes, we rely on a set of predefined rules coupled with the analytical capabilities of the tree diff technique. These predefined rules are designed to identify specific patterns that pertain to micro-changes, facilitating a targeted and efficient analysis.

The detection rule of the micro-change *Add Conjunct or Disjunct* is:

$$\text{belongs}(e, cond) \wedge \text{INS}(IEO.\&\&, e) \vee \text{INS}(IEO.||, e) \quad (1)$$

where e and $cond$ are AST nodes that represent expression and conditional expression in an if statement, respectively. The e, a child node of $cond$, is represented as $\text{belongs}(e, cond)$. The $IEO.\&\&$ and $IEO.||$ represent the infix expression operator, logical AND (&&), and infix expression operator, logical OR (||), respectively. The $\text{INS}(IEO.\&\&, e)$ indicates an insertion action of inserting the logical AND to the expression e. In other words, the rule can be expressed as within an existing conditional expression in an if statement, the insertion of a logical AND or logical OR operator is considered indicative of the *Add Conjunct or Disjunct* micro-change. This criterion is used to identify instances where an additional condition is appended to an existing conditional expression.

B. Catalog Creation

Our objective was to develop a comprehensive catalog of micro-changes targeting conditional-related changes, capturing a spectrum of these changes as broad as possible. Recognizing the significant time investment and effort required to build an exhaustive catalog, we adopted a systematic approach to manage the task within a feasible timeframe.

We began with the preliminary identification of 13 distinct micro-change types, characterized by the addition, deletion, and alteration of code elements within conditional expressions.

Then, to refine and expand our initial categorization, we selected the *mbassador* repository[6], which contains 226 conditional-related changes. The first author undertook three review cycles, focusing on changes not covered by our defined micro-change types.

During the first review, we discovered that 166 out of the total conditional changes fell outside of our catalog's scope.

We randomly picked a subset of 100 changes as a basis for enhancing the existing micro-change designs and for formulating additional new micro-change types. Following this refinement and a subsequent detection phase, the second review revealed 90 changes still not accommodated by our updated catalog. This prompted further enhancements and the introduction of new micro-change types, culminating in a catalog comprising 20 distinct types.

Upon implementing these revisions, a third review round was carried out, revealing that out of the 226 conditional-related change, 64 were still not covered by our micro-change catalog. However, these instances can all be attributed to incorrect parsing by GumTree, the tool we utilized for deriving tree diff. It occasionally fails to recognize the holistic structure of certain code modifications. For example, GumTree should recognize a change as removing a tree, where the root node is a conditional expression, and its children represent the associated code block. However, it occasionally misinterprets the change as removing a series of individual nodes, rather than recognizing the entire tree structure as a single conditional block removal.

Ultimately, our refined catalog achieved a 71.7% coverage rate of conditional-related changes, demonstrating substantial progress toward our goal of comprehensive coverage.

V. Empirical Study

A. Overview

To investigate micro-changes with respect to helping understand code changes, and to answer the research questions proposed in Section III-C, we conducted micro-change detection on a large-scale open-source Java repositories dataset.

[5]https://github.com/bennidi/mbassador/commit/744c029

[6]https://github.com/bennidi/mbassador

TABLE III
OVERVIEW OF DATASET COMPRISING 73 PROJECTS

Metric	Min	Q1	Median	Q3	Max	Total
Java files	39	425	814	2,067	10,012	114,080
Lines of code	4,200	33,681	64,048	166,517	970,310	10,931,743
Commits	342	2,025	3,588	7,810	14,971	366,697
Processed commits	254	1,732	3,298	6,179	14,605	312,554
Processed methods	208	7,341	17,143	40,085	178,437	2,051,128
Conditional-related commits	17	818	2,245	5,118	11,753	240,201
Conditional-related changes	36	1,925	5,609	13,504	33,222	637,669

To answer RQ_1, we calculate the coverage of micro-changes on conditional-related changes. We regard changes to if statements, their direct children, and any descendants of their conditional expressions as *conditional-related changes*. For RQ_2, the frequency of the micro-changes is calculated to uncover which type of micro-changes are commonly used.

B. Data Collection

The dataset we used is the one collected by Silva et al. [39], which contains 124 GitHub-hosted software projects. The selection of repositories for the dataset was governed by a set of criteria to ensure the relevance and feasibility of our analysis. Firstly, because the total number of commits of the repository is not a key feature influencing our investigation on micro-changes, we imposed a threshold on the number of commits, excluding any repositories with more than 15,000 commits to manage the complexity and execution time of our experiments. Secondly, we required that the repositories remain accessible via the links provided in the dataset curated by Silva et al., ensuring that our sources were both current and retrievable. Last, we excluded non-Java projects, as our current detection mechanism is specialized on Java source code. Employing these criteria, we composed a dataset comprising 73 repositories. The dataset is available in our supplementary package [40].

A plot of our dataset is shown in Fig. 4, where the x-axis is the number of commits being processed in our experiment, and the y-axis is the number of conditional-related changes.

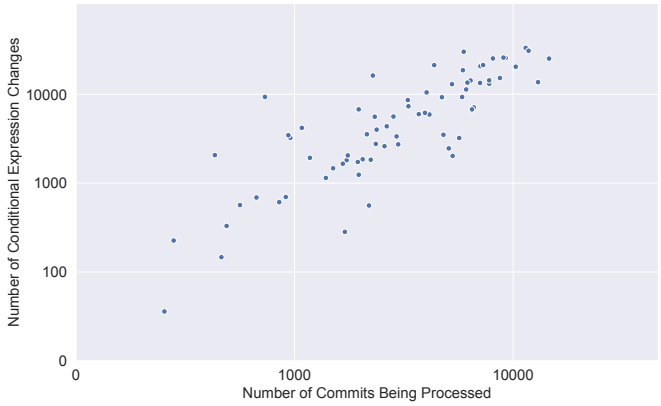

Fig. 4. Distribution of dataset.

Basically, the number of conditional-related changes increases with the number of commits being processed.

We also show an overview of our dataset in Table III, where the term *Processed commits* denotes the count of commits after excluding merge commits to avoid duplicate detection, excluding commits involving only additions or only deletions which is not the scope of this study. Similarly, the *Processed methods* pertains to the count of methods after removing those that have exclusively additions or deletions. In total, we processed 366,697 commits, where 240,201 of them feature changes affecting conditional expressions. The total number of conditional-related changes that we analyzed is 637,669.

C. RQ_1: To what extent can the designed micro-changes explain the code change?

1) Study Design: This research question aims to evaluate the breadth and depth with which our catalog of designed micro-changes can capture and accurately represent the essence of code changes. We implemented the micro-change detection methodology outlined in Section IV across our dataset to facilitate this analysis.

To quantitatively measure the outcome, we developed a metric measuring the coverage of conditional-related changes encompassed by our catalog of micro-changes.

As illustrated in Fig. 2, the annotations Ⓐ and Ⓓ denote lines of code changes identified as non-rename refactoring ($Ref^{\text{non-rename}}$) and rename refactoring (Ref^{rename}), respectively. Their union can capture the code changes involved with any refactorings detected by RefactoringMiner:

$$Ref = Ref^{\text{non-rename}} \cup Ref^{\text{rename}}. \tag{2}$$

Annotations Ⓑ and Ⓒ indicate the conditional-related changes identified through textual-diff ($Diff^{\text{text}}$) and tree-diff ($Diff^{\text{tree}}$), respectively, while Ⓔ highlights those condition-related changes captured by detected micro-changes in the catalog (MC).

The collective set of lines comprising conditional-related changes emerges from the intersection of $Diff^{\text{text}}$ and $Diff^{\text{tree}}$. This intersection leverages the strengths of both tree-based and textual diff algorithms to heighten the precision in pinpointing code alterations, mitigating the risk of falsely identifying unchanged code as changed. By integrating these approaches, the likelihood of false positives is reduced, thereby increasing

precision in the detection of genuine code changes. The set of conditional-related changes (*CRC*) is denoted as:

$$CRC = Diff^{\text{text}} \cap Diff^{\text{tree}}. \tag{3}$$

A subset of these conditional-related changes are addressed by our micro-changes.

The ratio quantifying micro-change coverage together with the detected refactorings (*Coverage*) is computed as:

$$Coverage = \frac{|(MC \cup Ref) \cap CRC|}{|CRC|}. \tag{4}$$

To underscore the unique contributions of micro-changes and traditional refactorings in capturing code changes, we introduce two distinct coverage ratios.

The *micro-change coverage* ($Coverage^{\text{MC}}$) focuses solely on the extent to which micro-changes from our catalog cover conditional-related changes, reflecting the specific impact of our catalog's micro-changes, which is denoted as:

$$Coverage^{\text{MC}} = \frac{|MC \cap CRC|}{|CRC|}. \tag{5}$$

The *refactoring coverage* ($Coverage^{\text{RM}}$) quantifies the extent of Rename and Non-Rename Refactorings identified by RefactoringMiner, showcasing the impact of traditional refactorings, which is denoted as:

$$Coverage^{\text{RM}} = \frac{|Ref \cap CRC|}{|CRC|}. \tag{6}$$

These metrics are crucial for clearly distinguishing the contributions of micro-changes from refactorings, ensuring a precise assessment of their roles in software development practices without overlap. The results of *Coverage*, $Coverage^{\text{MC}}$, and $Coverage^{\text{RM}}$ are compared to prove the effectiveness of our micro-change catalog in expressing code changes.

In addition, to validate the credibility of the coverage result, we randomly selected 100 detected micro-change instances from four repositories within the dataset. The first author of this paper manually reviewed those instances.

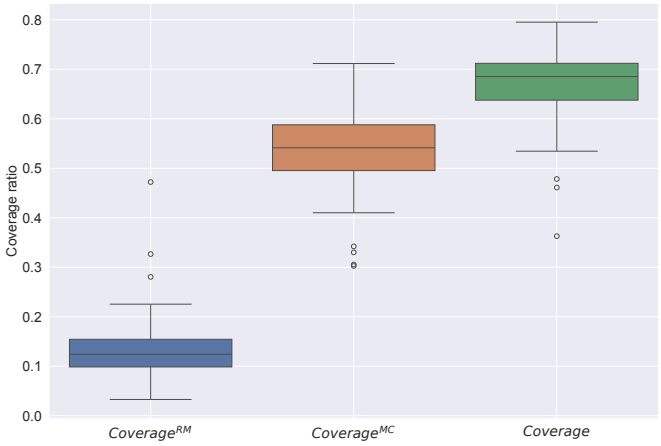

Fig. 5. Coverage across the dataset.

2) Results and Discussion: In general, the coverage of micro-changes on our dataset has an average ratio of 67.1%. The distribution is depicted in the boxplot in Fig. 5. The three boxes correspond to the $Coverage^{\text{RM}}$, $Coverage^{\text{MC}}$, and *Coverage*, indicating the coverage of RefactoringMiner, the coverage of micro-changes, and the combined impact of both. The medians for the three boxes are 0.124, 0.541, and 0.685, while the average values are 0.135, 0.536, and 0.671.

A comparative analysis between the $Coverage^{\text{RM}}$ and $Coverage^{\text{MC}}$ reveals a significant insight: There is a significant difference between these two, as evidenced by a paired t-test ($p < 0.001$) and Cohen's d of 5.5, which strongly suggests the difference is not by random chance. Our micro-change catalog achieves, on average, four times greater coverage of conditional-related changes than refactorings alone. This finding underscores the substantial efficacy of our catalog in capturing a broader spectrum of code changes, thereby affirming its value and precision. The coverage across each repository within our dataset of 73 repositories is detailed in Fig. 6.

With the exception of one repository *android-demo*, the remaining 72 repositories exhibit, on average, ca. 5 times greater $Coverage^{\text{MC}}$ than $Coverage^{\text{RM}}$. Notably, the repository *bassbox* has a $Coverage^{\text{MC}}$ of 0.444, which is 13 times greater than $Coverage^{\text{RM}}$, standing at 0.034.

Furthermore, an analysis reveals that 67.1% of the repositories possess $Coverage^{\text{RM}}$ of less than 0.15. This low coverage suggests that refactorings alone are insufficient in fully accounting for the essence of code changes within conditional expressions. In contrast, the average micro-change coverage ($Coverage^{\text{MC}}$) across these repositories stands at 0.56, which is approximately 3.7 times greater. This disparity indicates that the coverage achieved by combining both refactorings and micro-changes (*Coverage*) is largely attributed to the micro-changes that we have cataloged. The micro-changes in our catalog thus serve as a complementary extension to refactorings, significantly enhancing the ability to explain conditional-related changes.

To ensure the correctness, the first author manually reviewed 100 detected micro-changes randomly extracted from four repositories. The review revealed that 86% of these instances were true positives, underscoring the reliability of our detection methodology.

The causes of false positives can be divided into two: 1) inaccuracies in diff parsing by GumTree, and 2) instances where a single change undergoes multiple micro-changes, making it impossible to capture a change accurately with just one micro-change.

The first issue underscores a clear opportunity for enhancement within our detection methodology. Employing a more precise tree-diff tool could markedly increase the detection accuracy, pointing to a vital area for technical refinement.

The second issue introduces the concept of *compound micro-change*, where multiple micro-changes are applied to a single code element. An example is shown in Fig. 7. It is a

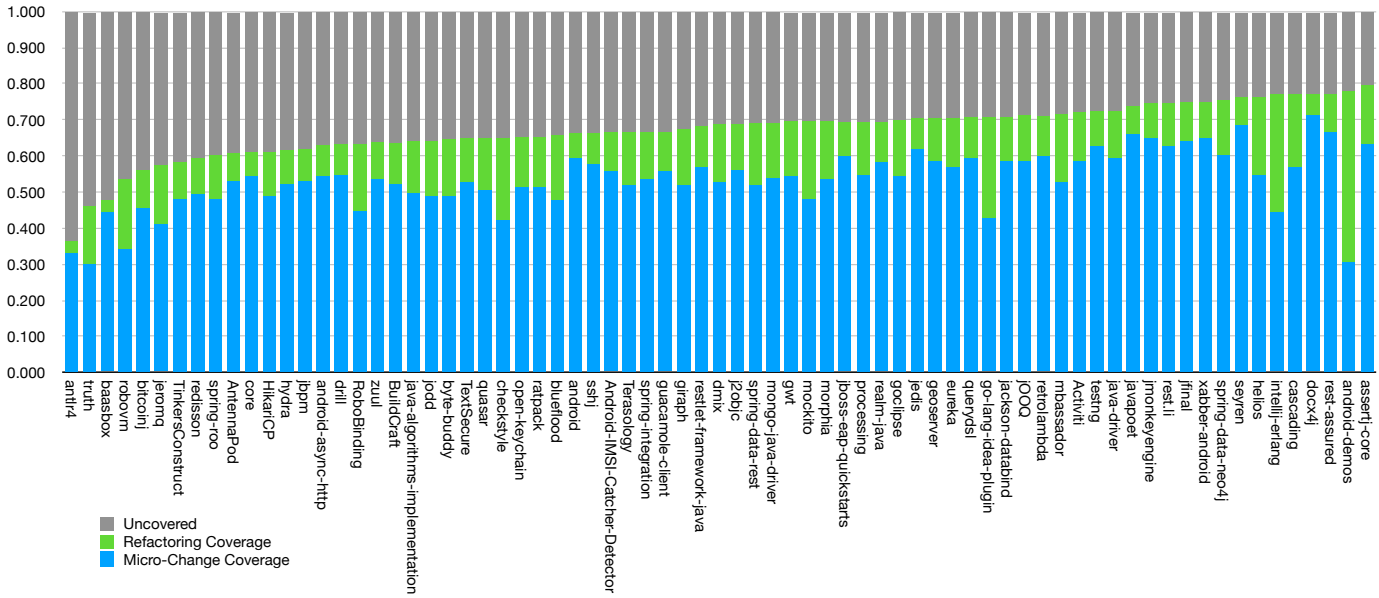

Fig. 6. Coverage across all 73 repositories, sorted according to the sum of Refactoring coverage and Micro-Change Coverage.

code change sourced from the repository *retrolambda*[7]. Here, the code modification involves three distinct micro-changes: *Remove Conjunct or Disjunct*, *Extract Variable* and *Reverse Condition*. This change is too complicated for tree diff to comprehensively parse the sequence of actions. As a result, only the *Remove Conjunct or Disjunct* is detected, with the additional code being misclassified as a new insertion (*Add Conditional Statement*) due to its complexity.

Additionally, we examined changes in conditional expressions that fell outside the scope of both our micro-change catalog and refactorings. The same two issues that contributed to false positives were also responsible for these uncovered changes. Furthermore, though we did not observe in our review process, we do not deny the possibility that certain changes possess the potential to be classified as new types of micro-changes, indicating an opportunity for expanding our catalog to encompass a broader spectrum of modifications.

> On average, micro-changes can explain 67.1% of the conditional-related changes. Micro-changes in the catalog have a better ability to explain changes than refactorings. The detection mechanism achieves an accuracy of 86%.

D. RQ_2: What are the frequencies of the micro-changes types?

1) Study Design: In this research question, we calculate the detected frequency of the micro-changes designed in our catalog in the dataset. In addition, we also deduce the developers' motivation for introducing the most frequent micro-change type in the development.

The set of micro-changes detected from the dataset of type $t \in T$ is denoted as MC_t, where T is the set of the types

[7]https://github.com/luontola/retrolambda/commit/e3c8647

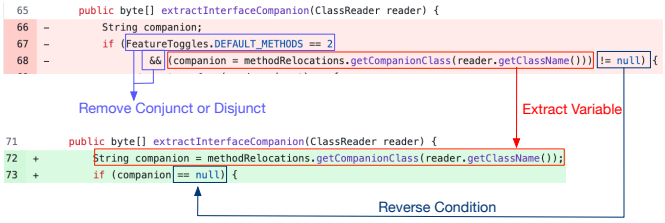

Fig. 7. Multiple micro-changes in a single conditional expression.

in our catalog. The frequency of a micro-change of type t is calculated as:

$$freq(t) = \frac{|MC_t|}{\sum_{t' \in T} |MC_{t'}|}. \quad (7)$$

2) Results and Discussion: The frequencies of each type of micro-changes are detailed in Table IV. For a given micro-change type t, the table lists the number of detected instances in the dataset, the $freq(t)$, along with their respective rank.

The highest frequency micro-change type is *Add Conditional Statement*, exhibiting a frequency of 28.1%, whereas the *Adjust Condition Boundary* is the least frequent, with a frequency of 0.1%. The second most frequent type is *Wrap Statement in Block*, indicating a prevalent trend towards discarding syntactic sugar in Java, which allows omitting curly braces surrounding a single statement at a then/else clause. The third most frequent type, *Remove Conditional Statement*, in conjunction with the most frequent *Add Conditional Statement*, implies that modifications involving the introduction or removal of entire conditional blocks are more common than minor adjustments, such as changing the logic operators or boundary conditions. Notably, both changes are *not* behavior-preserving, i.e., they represent actual changes in the program logic.

TABLE IV
FREQUENCY OF MICRO-CHANGES DETECTED

Micro-change	Occurrences	Frequency	Rank
Add Conditional Statement	2,931,551	28.10%	1
Wrap Statement in Block	2,569,619	24.60%	2
Remove Conditional Statement	1,245,424	11.90%	3
Unwrap Statement from Conditional	954,654	9.20%	4
Wrap Statement in Conditional	919,582	8.80%	5
Remove Conjunct or Disjunct	501,002	4.80%	6
Reverse Condition	362,179	3.50%	7
Add Conjunct or Disjunct	316,263	3.00%	8
Extend Else with If	208,262	2.00%	9
Unwrap Statement from Block	142,066	1.40%	10
Move inward Condition	56,305	0.50%	11
Remove Else	42,201	0.40%	12
Conditional to Boolean Return	38,046	0.40%	13
Extend If with Else	37,886	0.40%	14
Flip Logic Operator	29,949	0.30%	15
Conditional to Expression	25,475	0.20%	16
Move outward Condition	19,809	0.20%	17
Conditional to Switch	18,640	0.20%	18
Swap Then and Else	8,167	0.10%	19
Adjust Condition Boundary	6,004	0.10%	20
Total	**10,433,084**		

Furthermore, the *Add Conjunct or Disjunct* and *Remove Conjunct or Disjunct* occupy the 8th and 6th ranks, respectively. This indicates a relatively high occurrence of refinements in conditional expressions, either by incorporating new conditions or simplifying them. Conversely, the micro-change types *Conditional to Boolean Return*, *Conditional to Switch*, and *Conditional to Expression* rank 13th, 18th and 16th, respectively, indicating their low prevalence. This could be attributed to the less frequent application of *Boolean Return*, *Switch*, and *Conditional Operator* compared to other more common patterns.

For the most frequent micro-change type *Add Conditional Statement*, we categorized the motivations of developers to introduce this micro-change into three types:

1) **Security/Compatibility Check**: This type of conditional statement addition emerges from ensuring that only users or systems meet specific security criteria or have compatible resources that can execute the operations. An example is found in repository *retrolambda*[8] and illustrated in Fig. 8. The added code block is to ensure the program only operates on Java 8, preventing runtime errors or unexpected behavior on incompatible Java versions.

```
14      public static void main(String[] args) {
15          System.out.println("Retrolambda " + getVersion());
16
17  +       if (!isRunningJava8()) {
18  +           System.out.println("Error! Not running under Java 8");
19  +           System.exit(1);
20  +       }
21  +
22          Config config = new Config(System.getProperties());
```

Fig. 8. Conditional statement added for security check.

[8]https://github.com/luontola/retrolambda/commit/2501461

2) **Performance Optimization**: The motivation for this type is more about performance, which is to check for and utilize optimal configurations or system conditions that enhance the program's performance. Depicted in Fig. 9, the example found in repository *retrolambda*[9], as commented by the developer, the code returns earlier on classes whose names start with *"java/"* to avoid continue processing the unnecessary operations.

```
24      public void analyze(byte[] bytecode) {
25          ClassReader cr = new ClassReader(bytecode);
26  +       String className = cr.getClassName();
27  +       if (className.startsWith("java/")) {
28  +           // the JVM disallows user classes in java.* packages, so don't even try
        backporting them
29  +           return;
30  +       }
31  +       Type clazz = classNameToType(className);
32
```

Fig. 9. Conditional statement added for performance optimization.

3) **User Assistance**: This type of insertion will not affect the functionality of the program, and it is to provide feedback or help to the user if the program's prerequisites are not met, enhancing user experience or improving the user's ability to resolve configuration issues more efficiently. An example is found in repository *zuul*[10] and shown in Fig. 10. By checking the format of the input and logging an error if it does not meet the expected format, the system provides immediate feedback that can help the user or developer identify and correct the issue.

```
74                  for (String hostAndPort : hostAndPorts) {
75                      String[] split = hostAndPort.split(":");
76  +                   if (split.length != 2) {
77  +                       LOG.error("Each service should be in format <host>:<port>!
        hostAndPort=" + hostAndPort);
78  +                   }
79                      servers.add(new ServerInfo(split[0], Integer.parseInt(split[1])));
```

Fig. 10. Conditional statement added for user assistance.

We reported the frequency of each type of designed micro-change in the catalog. The most frequent types are *Add Conditional Statement*, *Wrap Statement in Block*, and *Remove Conditional Statement*, which together account for nearly 2/3 of the micro-changes.

VI. THREATS TO VALIDITY

Internal Validity: The concept of micro-change is defined broadly and abstractly, which also enables us to define various trivial and uninteresting types, according to the definition of the micro-change, e.g., addition or removal of a specific character. We leave open the discussion about the boundary between micro-changes and non-micro-changes. Indeed, there is still a lot of beneficial vocabulary to describe changes that are not covered by the well-known existing change vocabulary, such as refactoring operations. Another possible threat is

[9]https://github.com/luontola/retrolambda/commit/281ef93
[10]https://github.com/Netflix/zuul/commit/040d3ef

that we excluded the renaming of methods as introduced in Section IV-A. We could treat the renamed method as the same file and examine its differences, rather than just the renaming itself. While this is feasible, it adds complexity. To simplify our analysis, we chose to ignore renaming.

Construct Validity: The comprehensiveness of our conditional-related micro-change catalog is critical for the validity of our findings regarding the scope of code change it can explain. There is a possibility that our catalog may not encompass all relevant micro-change types, especially given the vast and evolving nature of software development practices. As a result, our analysis and conclusions about how common and important different types of micro-changes might not be completely accurate. However, adding more micro-change types to the catalog does not impact the existing ones, but rather lowers even more the changes unaccounted for.

External Validity: Our selection criteria for repositories, specifically those with fewer than 15,000 commits, may limit the generalizability of our findings. This threshold was chosen to manage the scale and complexity of the analysis, which took several days. We cannot exclude that larger repositories could exhibit different patterns of micro-changes or refactoring activities. Our results may therefore not fully represent the diversity of software development practices in projects of significantly larger scale, potentially affecting the applicability of our conclusions to such contexts.

VII. Reflection

The concept of micro-change, as explored in this study, opens up novel avenues for enhancing software development and maintenance practices. Two particularly promising applications of micro-change that we envisage are in the realms of commit message generation and the development of a tool that not only highlights code changes but also annotates them with corresponding micro-change types.

Commit Message Generation: An automated commit message generation system powered by the concept of micro-changes could significantly improve the efficiency of the version control workflow. By analyzing the micro-changes within a commit, such a system could generate concise, descriptive commit messages that accurately reflect the essence of the code changes. This would not only save developers and reviewers' time for understanding code change but also improve the quality of documentation within repositories, making it easier to understand the history and intent of changes. Such messages could enhance clarity for future maintenance, reviews, and collaborative work by providing a clear and automatically generated summary of the impact of each commit.

Micro-Change Annotation System: We think that it is possible to build a development tool that goes beyond simply highlighting differences between code versions. Such a tool would leverage the micro-change catalog to annotate changes with specific micro-change types, offering developers immediate insights into the nature of each modification. For instance, a change categorized under *Reverse Condition* would be clearly marked as such, providing context at a glance. This could facilitate a more nuanced understanding of code evolution for developers reviewing changes, thus improving code review efficiency and accuracy.

Moreover, by making the implications of changes more transparent, such a tool could aid in educational contexts, helping new developers understand coding practices and patterns through real examples. Both these applications highlight the potential of micro-change to not only advance current development practices but also to foster a deeper understanding of code evolution. As we continue to refine and complete our micro-change catalog and detection methodologies, we look forward to exploring these and other applications of micro-changes.

VIII. Conclusion and Future Work

In this paper, we introduced the concept of micro-changes, transforming code diffs into a series of predefined operations described in natural language. We have developed a catalog tailored to conditional-related micro-changes, complete with specific detection rules. Our detection efforts across 73 open-source repositories reveal that, on average, 67.1% of conditional-related changes are covered by micro-changes.

In the future, we aim to broaden the scope of our catalog beyond conditional-related changes and explore practical applications for leveraging micro-changes.

Acknowledgments

This work was partly supported by the Young Researchers Exchange Programme between Japan and Switzerland under the Japanese-Swiss Science and Technology Programme, JST SPRING No. JPMJSP2106, and JSPS Grants-in-Aid for Scientific Research Nos. JP24H00692, JP23K24823, JP21H04877, JP21K18302, and JP21KK0179.

References

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, 2013, pp. 712–721.

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2018)*, 2018, pp. 181–190.

[3] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-aware configuration management," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009, pp. 123–134.

[4] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE 2009)*, 2009, pp. 309–319.

[5] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004, pp. 210–219.

[6] M. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[7] A. von Mayrhauser and A. M. Vans, "From code understanding needs to reverse engineering tool capabilities," in *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (IWCASE 1993)*, 1993, pp. 230–239.

[8] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. C. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring neural efficiency of program comprehension," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE 2017)*, 2017, pp. 140–150.

[9] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "A look into programmers' heads," *IEEE Transactions on Software Engineering*, vol. 46, no. 4, pp. 442–462, 2020.

[10] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu, "What makes a good commit message?" in *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, 2022, pp. 2389–2401.

[11] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proceeding of a Symposium on Software Validation: inspection-testing-verification-alternatives*, 1984, pp. 13–40.

[12] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: An effective verification process," *IEEE Software*, vol. 6, no. 3, pp. 31–36, 1989.

[13] 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 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*, 2021, pp. 348–357.

[14] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[15] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[16] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[17] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in *Proceddings of the 4th International Workshop on Mining Software Repositories (MSR 2007)*, 2007.

[18] ——, "Tracking your changes: A language-independent approach," *IEEE Software*, vol. 26, no. 1, pp. 50–57, 2008.

[19] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, "LHDiff: A language-independent hybrid approach for tracking source code lines," in *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*, 2013, pp. 230–239.

[20] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*, 2002, pp. 380–388.

[21] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[22] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, 2014, pp. 313–324.

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

[24] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois Urbana-Champaign, 1992.

[25] J. Chen, J. Xiao, Q. Wang, L. J. Osterweil, and M. Li, "Perspectives on refactoring planning and practice: an empirical study," *Empirical Software Engineering*, vol. 21, pp. 1397–1436, 2016.

[26] D. Silva and M. T. Valente, "RefDiff: Detecting refactorings in version histories," in *Proceedings of the 14th IEEE/ACM International Conference on Mining Software Repositories (MSR 2017)*, 2017, pp. 269–279.

[27] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

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

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

[30] F. Coelho, T. Massoni, and E. L. 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.

[31] 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.

[32] S. Thangthumachit, S. Hayashi, and M. Saeki, "Understanding source code differences by separating refactoring effects," in *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC 2011)*, 2011, pp. 339–347.

[33] X. Ge, S. Sarkar, and E. Murphy-Hill, "Towards refactoring-aware code review," in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2014)*, 2014, pp. 99–102.

[34] E. L. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, 2014, pp. 751–754.

[35] 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.

[36] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking Java methods with Git mechanisms," *Journal of Systems and Software*, vol. 165, pp. 110 571:1–13, 2020.

[37] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for Java," in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*, 2011, pp. 96–100.

[38] S. Shiba and S. Hayashi, "Historinc: A repository transformation tool for fine-grained history tracking," *Computer Software*, vol. 39, no. 4, pp. 75–85, 2022.

[39] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 2016, pp. 858–870.

[40] L. Chen, M. Lanza, and S. Hayashi, "Supplementary package for understanding code change with micro-changes," https://doi.org/10.6084/m9.figshare.25592220, 2024.