PROGRAM INSPECTION AND TESTING TECHNIQUES FOR CODE CLONES AND REFACTORINGS IN EVOLVING SOFTWARE

Zhiyuan Chen

University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Please take our feedback survey at: https://unomaha.az1.qualtrics.com/jfe/form/SV_8cchtFmpDyGfBLE

Recommended Citation


https://digitalcommons.unomaha.edu/studentwork/2912

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
PROGRAM INSPECTION AND TESTING TECHNIQUES FOR CODE CLONES AND REFACTORINGS IN EVOLVING SOFTWARE

A Thesis
Presented to the
Department of Computer Science
and the
Faculty of the Graduate College
University of Nebraska
In Partial Fulfillment
of Requirements for the Degree
Master of Science
University of Nebraska at Omaha
by
Zhiyuan Chen

October 2017

Supervisory Committee:
Dr. Myoungkyu Song
Dr. Harvey P. Siy
Dr. Haorong Li
ABSTRACT OF THE THESIS

Program Inspection and Testing Techniques for Code Clones and Refactorings in Evolving Software

Zhiyuan Chen, MS
University of Nebraska, 2017
Advisor: Dr. Myoungkyu Song

Developers often perform copy-and-paste activities. This practice causes the similar code fragment (aka code clones) to be scattered throughout a code base. Refactoring for clone removal is beneficial, preventing clones from having negative effects on software quality, such as hidden bug propagation and unintentional inconsistent changes. However, recent research has provided evidence that factoring out clones does not always reduce the risk of introducing defects, and it is often difficult or impossible to remove clones using standard refactoring techniques. To investigate which or how clones can be refactored, developers typically spend a significant amount of their time managing individual clone instances or clone groups scattered across a large code base.

To address the problem, this research proposes two techniques to inspect and validate refactoring changes. First, we propose a technique for managing clone refactorings, Pattern-based clone Refactoring Inspection (PRI), using refactoring pattern templates. By matching the refactoring pattern templates against a code base, it summarizes refactoring changes of clones, and detects the clone instances not consistently factored out as potential anomalies. Second, we propose Refactoring Investigation and Testing technique, called RIT. RIT improves the testing efficiency for validating refactoring changes. RIT uses PRI to identify refactorings by analyzing original and edited versions of a program. It then uses the semantic impact of a set of identified refactoring changes to detect tests whose behavior may have been affected and modified by refactoring edits. Given each failed assert, RIT helps developers focus their attention on logically related program statements by applying program slicing for minimizing each test. For debugging purposes, RIT determines specific failure-inducing refactoring edits, separating from other changes that only affect other asserts or tests.

Keywords: regression testing, code clones, refactorings, program differencing, code change analysis.
# TABLE OF CONTENTS

1 INTRODUCTION ................................................. 1
   1.1 Research Agenda ........................................ 2
   1.2 Major Research Contribution ................................. 3
      1.2.1 Summarization for Clone Refactorings (RQ1) ........ 3
      1.2.2 Detection for Incomplete Clone Refactorings (RQ2) ... 4
      1.2.3 Identification for Test-Slices Affected by Refactorings (RQ3) 4
      1.2.4 Detection for Failure-Inducing Refactorings (RQ4) ....... 4
   1.3 Outline .................................................. 4

2 RELATED WORK ............................................................ 5
   2.1 Code Search and Inconsistency Detection ................. 5
   2.2 Clone Detection ............................................. 5
   2.3 Refactoring Identification .................................. 6
   2.4 Manual Refactorings ......................................... 7
   2.5 Change Impact Analysis ..................................... 8
   2.6 Fault Localization .......................................... 8

3 PATTERN-BASED CLONE REFACTORING INSPECTION ................. 9
   3.1 Motivating Example ......................................... 9
   3.2 Approach .................................................. 11
      3.2.1 Phase I: Change Summarization for Clone Refactorings ... 12
      3.2.2 Phase II: Detecting Incomplete Clone Refactorings .... 17
      3.2.3 Phase III: Visualizing Clone Refactorings and Anomalies ... 19

4 REFACTORINGS INVESTIGATION AND TESTING TECHNIQUE ............ 20
   4.1 Motivating Example ......................................... 20
   4.2 Approach .................................................. 23
      4.2.1 Change Category ........................................ 24
**List of Figures**

3.1 A simplified example: A clone group is refactored inconsistently. Cloned regions are highlighted, deleted code is marked with ‘-’ and added code marked with ‘+’ .................................................. 10

3.2 A viewer in PRI that shows clone refactoring summarization and refactoring anomaly detection regarding Figure 3.1 ........................................... 11

3.3 Overview of PRI’s workflow .................................................. 12

3.4 Extracting the edit operations from clone changes .......................... 13

3.5 Detecting refactoring anomalies in clone refactoring evolution ........... 18

3.6 Visualizing clone refactorings and anomalies regarding Extract Method (see more screenshots at http://faculty.ist.unomaha.edu/msong/pri/) .............................. 18

4.1 An example: three clone groups are refactored, some of which cause test failures. Cloned regions are highlighted: the first clone group is highlighted in red, the second one in blue, and the last one in green. .................. 21

4.2 Regression tests test1() and test2() for testing an example program in Figure 4.1 ........................................................................................................ 22

4.3 Generating subtests each of which address an independent development and maintenance concern .................................................. 23

4.4 The system overview of RIT .......................... 24

4.5 The refactoring change dependence graphs .................................. 25

4.6 The call graphs for tests before and after the refactorings, representing (1) methods by rectangles, (2) fields by rectangles with dashed line, (3) call and field-access edges by directed lines, and (4) associated labels on edges by green rectangle. The partitioned test-slices are indicated with dotted-gray boxes .................................................. 25
4.7 Applying data flow and aliases analyses to a test-slice \texttt{test2-2()} to analyze semantic dependencies between procedures using inter-procedural data flow information such as alias. $S_n$ denotes a sequence of statements on control flow graph nodes; a dotted-line a backward analysis direction; a solid line a forward analysis direction; a underlined statement a location identified by an aliases analysis, and; a dotted-circle a correspondence of propagated $v_t$. Despite an unperformed analysis within method \texttt{A.m3(F)} skipped by a preceding analysis result, the access of field variables \texttt{this.i} and \texttt{this.k} can be determined by our approach.

5.1 A false negative example from the Apache Tomcat (r1042872) project (the regions with highlighted background are cloned).
# List of Tables

3.1 Clone refactoring templates that PRI supports by tracking the evolution of clones focusing on their removal refactorings.  
3.2 The six types of classification that PRI annotates classified clone instances with symbols in the first column s.  
5.1 Subject applications (File: the number of files, LOC: lines of code, and Cor: count of refactorings)  
5.2 Accuracy of PRI’s summarization and detection. RFT: the refactoring types that developers perform across VER revisions (see Table 3.1 for acronyms), VER: the number of revisions where developers apply refactorings, Tim: the time that PRI completes each task (an average of time (sec.) per group), Cls: the number of refactored clones correctly summarized by PRI (instance/group), Gt: the number of the ground truth data set for clone refactoring summarization (instance/group), P: precision (%), R: recall (%), and A: accuracy (%), and each line represents the evaluation result for a project at a particular revision where developers apply clone refactoring applications.  
5.3 Accuracy of PRI’s summarization and detection. Rft: the refactoring types that developers perform across revisions, Cls: the number of unrefactored clones correctly detected by PRI (instance/group), Gt: the number of the ground truth data set for incomplete clone refactoring detection (instance/group), X, Xl, Xm, Xo, and Xs: see Table 3.2, P: precision (%), R: recall (%), and A: accuracy (%), and each line represents the evaluation result for a project at a particular revision where developers apply clone refactoring applications.  
5.4 The categories of seeded refactoring anomalies, consisting of four major categories such as ME (20%), EE (40%), LCa (20%), and RCa (20%)
5.5 Accuracy of RIT’s capability to identify affected test-slices and detect failure-inducing refactorings. PRJ an evaluated project at a particular revision (L for Log4j, J for Jruby and T for Tomcat), AT, the number of affected failed tests in ground truth, AS, the number of affected failed test-slices in ground truth, AT: the number of affected failed tests identified by RIT, AS: the number of affected failed test-slices identified by RIT, AS: the number of total test-slices identified by RIT, AS: the number of affected failed test-slices correctly identified by RIT, AS: the number of unrelated test-slices of failed test(s), P1: precision of affected test identification(%), R1: recall of affected test identification(%), A1: accuracy of affected test identification(%), and each line represents the evaluation result for a project at a particular revision where manual refactoring changes have been conducted.

5.6 Accuracy of RIT’s capability to identify affected test-slices and detect failure-inducing refactorings. PRJ an evaluated project at a particular revision (L for Log4j, J for Jruby and T for Tomcat), AR, the number of affecting refactorings in ground truth, AR: the number of affecting refactorings identified by RIT, AR: the number of affecting unique refactorings correctly identified by RIT, AR: the number of affecting refactorings correctly identified by RIT, AR: the number of non-affecting refactorings, P2: precision of affecting refactoring identification(%), R2: recall of affecting refactoring identification(%), and A2: accuracy of affecting refactoring identification(%), and each line represents the evaluation result for a project at a particular revision where manual refactoring changes have been conducted.
CHAPTER 1

INTRODUCTION

Code changes are usually repetitive [25, 54]. Recent research has pointed out that over 30% of the total amount of code is repetitive regarding various application domains, such as operating systems, web server programs and development environments [36, 54, 65, 69]. Changing the code similarly is an easy way to achieve a design goal in adding new features and fixing numerous bugs, mostly because of the copy-paste(-and-adapt) programming practice, the framework-based development, and the reuse of the same design patterns or libraries, thus creating code clones.

While several studies [40, 41, 22] find positive aspects of code clones, in other studies [30, 3, 49], cloning is considered harmful to software quality, leading to much effort on detection and removal of code clones. For example, anomalous changes could repeat either by a developer’s own error or by other developers’ fault unknowingly [3]. Code clones also require that changes to one section of code clones are to be propagated to multiple locations consistently, incurring additional maintenance costs to synchronize cloned regions [30, 49].

Despite high performance in detecting code clones [66], understanding clone groups (i.e., sets consisting of two or more clone instances) remains challenging. To improve understandability and maintainability, clone management tools have been developed. These techniques represent differences across multiple clone instances [45], track evolving code clones in a repository on a Version Control System (VCS) [14], and assist changes of clones and their contexts [28].

In management of code clones, developers often conduct clone refactoring, combining regions of code that are very similar and moving them into a function without altering the existing functionality. To help developers conduct refactorings for clone removal, Integrated Development Environments (IDEs), such as Eclipse, provide automated refactoring features. Unfortunately, not all developers make consistent use of these features, since refactorings are not always feasible by standard refactoring engines in IDEs [36]. Professional developers also tend not to use automated refactoring despite their awareness of the refactoring features of IDEs [77]. These manual refac-
torings often lead to error-proneness. Recent studies [34, 58] report that the ratio of refactorings affects increasing numbers of bugs, and Park et al. [58] find that regarding an omission error type such as incomplete refactorings, it takes much longer to be resolved than other types of defects. As a result, it is difficult and time-consuming for code reviewers to inspect individual or a group of clone instances and answer questions such as “How should these clones be refactored completely or correctly?” and “Are there any clones that should be removed along with these clones?”.

This research proposes two techniques for inspecting and validating refactoring changes. First, we propose a technique for inspecting refactorings of evolving clones and detecting incomplete refactorings: Pattern-based clone Refactoring Inspection (PRI). To examine which or how clones are refactored and to determine whether clones are refactorable, we implemented refactoring pattern templates, which are automatic Abstract Syntax Tree (AST) matching programs based on standard refactoring types from Fowler’s catalog [17]. We target five refactoring types: Extract Method, Pull Up Method, Move Method, Extract Super Class, and Move Type To New File, and two composite refactorings: Extract & Move Method and Extract & Pull Up Method.

Second, we propose Refactoring Investigation and Testing technique, called RIT which improves the efficiency of refactoring change validation. RIT is used as follows. RIT analyzes source code edits between the original and edited versions of a program’s AST to detect refactorings. Given a suite of regression tests, it then applies a change impact analysis technique to determine tests whose behaviors are potentially affected by the changes in refactorings. To reduce the cost of regression testing to run affected tests, RIT utilizes program slicing and data flow analysis techniques [78] to only identify semantically dependent statements for executing a failed assert statement in each test. To reduce the time and effort spent in debugging, RIT determines specific failure-inducing refactoring edits that are responsible for a given test’s failure.

1.1 Research Agenda

In this research, we address the following research questions in light of the challenges of testing and code review in the process of validating and comprehending an extensively refactored program.

- **RQ1**: Can PRI accurately summarize clone refactorings?
  
  We evaluate the accuracy of PRI’s clone refactoring summarization. We manually construct the ground truth of refactoring changes to clones on a sampled
data set from open source projects, such as AlgoUML, Apache Tomcat, Apache Log4j, Eclipse AspectJ, JEdit, and JRuby. The comparison between the PRI’s results against this ground truth shows how it demonstrates the summarization capability.

- **RQ2**: Can PRI accurately detect incomplete clone refactorings?
  We apply PRI to a data set with incomplete refactorings of clones performed by real developers. We assess how accurately PRI can detect such clone refactoring anomalies by automatically tracking clones and the corresponding refactorings across revisions.

- **RQ3**: Can RIT accurately determine test-slices affected by atomic changes of refactorings?
  We evaluate the accuracy of RIT’s affected test detection. We apply it to a data set composed by over 100 refactoring transformation on a sampled data set from open source projects. The comparison between the RIT’s results against this ground truth demonstrates how effectively it detects tests whose behaviors are potentially affected by refactoring changes.

- **RQ4**: Can RIT accurately detect affecting refactorings that cause the failure of these test-slices?
  We apply RIT to a data set with seeded refactoring anomalies, which do not produce compilation errors. We measure how accurately RIT can detect such refactoring anomalies (i.e., failure-inducing refactoring edits), and thus can reduce the amount of time and effort spent in debugging the changes responsible for a given test’s failure.

### 1.2 Major Research Contribution

#### 1.2.1 Summarization for Clone Refactorings (RQ1)

We propose a new approach for inspecting clones for refactoring. PRI provides a novel integration of program differencing and AST-based code pattern search to track the changes to clones based on well-known clone removal refactorings [50, 77].
1.2.2 Detection for Incomplete Clone Refactorings (RQ2)

PRI also detects unrefactored clones, extracts clone differences, and classifies these clones which either co-evolved, diverged, or remain unchanged. It extracts clone differences in the same clone group to provide a related reason of not being refactorable.

1.2.3 Identification for Test-Slices Affected by Refactorings (RQ3)

We propose a novel approach for inspecting and validating changes in manual refactorings [50, 37, 53, 77, 52] based on change impact analysis. RIT detects well-known refactorings [50, 77] by applying program differencing and AST-based code pattern search techniques. Given refactorings, it applies change impact analysis to determine tests that are affected by such changes.

1.2.4 Detection for Failure-Inducing Refactorings (RQ4)

RIT also identifies a subset of changes responsible for these anomalies, if a test fails due to refactoring anomalies. We develop a heuristic-based approach to improve the atomicity of composite tests. Given program dependence graphs by program slicing, RIT partitions a composite test and identifies semantically related statements dependent on each assert, merging these statements isolated from others. By applying data flow tracking, each partitioned test is more cohesive and self-contained with respect to the issue being addressed.

1.3 Outline

The rest of this research is organized as follows. Chapter 2 surveys related work. This can be divided into six main categories: Clone Detection, Refactoring Identification, Manual Refactoring, and Change Impact Analysis, and Fault Localization. Chapter 3 shows Pattern-based clone Refactoring Inspection (PRI). Chapter 4 describes another proposed technique Refactoring Investigation and Testing technique, called RIT. Chapter 5 outlines how we assess our approach. Chapter 6 presents future work and conclusions.
CHAPTER 2

RELATED WORK

2.1 Code Search and Inconsistency Detection

Several approaches detect inconsistencies in clones. CP-Miner [44], SecureSync [59] and Jiang et al.’s technique [30] reveal clone-related bugs by finding recurring vulnerable code. CBCD reveals bug propagation in clones by finding similar ASTs in a program dependence graph [43]. SPA analyzes discrepancies in changes and detects inconsistent updates in clones [61].

Our approach differs from these inconsistency detection techniques in two ways. First, PRI automatically accesses to VCS and incrementally identifies inconsistent changes in clone histories, unlike analysis of one or two versions. Due to these differences, in our case studies, we could not directly compare with existing clone-based code search techniques since these tools are not designed for inspecting clone evolution and its refactoring edits. Second, in contrast to PRI’s refactoring classification, the clones found by these tools require manual inspection to determine if these clones can be removed using standard refactoring techniques [17].

2.2 Clone Detection

Göde and Koshke [23] present an incremental clone detection algorithm to study clone evolution and find that most clones remain unchanged during their lifetime, and clones are mostly changed inconsistently. Krinke’s [39, 40] study investigates the changes to clones in five open source systems and find that some clone groups are changed consistently. Saha et al. [68] investigate the evolution of clones, and their results show that clone type is more likely to change inconsistently, when clones form gaps among clone fragments. Aversano et al. [2] study how clones are evolved and find that developers almost propagate the change consistently. Bettenburg et al. [8] investigate the effect of inconsistent changes to code clones and observe that 1–3% of inconsistent changes to clones introduce defects, reporting that most of clones are consistently maintained.
Although these approaches map clones between revisions of a program, they do not provide summarization of changes to clones for investigating clone refactorings.

These studies show that removing code clones is not always necessary nor beneficial. PRI helps to summarize clone evolution and refactorings, which developers can investigate during peer code reviews.

Toomim et al. [75], Duala-Ekoko and Robillard [14], Hou et al. [28] and Nguyen et al. [55] manage clones in evolving software by tracking the changes in the clones as code evolves. Lin et al. [45] also design a plug-in built on Eclipse IDE that computes differences among clones and highlights the syntactic differences across multiple clone instances. Unlike the above approaches, PRI leverages the clone region information to help code reviewers detect incomplete refactorings of clone groups which are omission-prone, supporting clone-aware refactoring recommendations.

2.3 Refactoring Identification

Tsantalis et al. [76] use a program slicing technique to capture code modifying an object state and design rules to identify refactoring candidates from slices. Bavota et al.’s [7] approach identifies classes to extract by using similarity and dependence between methods in a class. While these tools identify refactoring opportunities for clones, they do not support developers with real refactoring examples. Our approach provides concrete information of clone differences showing real refactoring examples of other siblings in the group.

RefFinder [60] could be used to search for refactorings. However, we find that it includes false negative refactoring cases such as clones that were factored out into nested method calls. In contrast to analysis of only VCS data in RefFinder, PRI analyzes both clone groups in clone database and source code in VCS to capture more precise data for identifying clone refactorings.

BeneFactor [19] and WitchDoctor [16] use refactoring patterns to help developers complete refactorings that were started manually. PRI uses refactoring patterns, but these patterns are automatically matched with clones across revisions.

Kim et al. [36] study the evolution of clones and provide a manual classification for evolving code clones. Unlike their approach, we automatically classify clones if they are not easily refactorable using standard refactoring techniques [17].
2.4 Manual Refactorings

Opdyke and Johnson [56] create the term refactoring, and later Fowler [17] presents a catalog of 72 different refactoring types that describes mechanical procedures in terms of required edits.

Major IDEs such as Eclipse, IntelliJ IDEA, and Visual Studio provide automated tools for refactorings. Ideally, a developer will always use a refactoring tool if one is available, since those tools check pre- and post-conditions to prevent refactoring anomalies. However, recent studies find that most developers underuse automated refactoring tools. Murphy et al. inspect commits in a repository and usages of the refactoring tool, and then correlate the refactoring in terms of the application of a refactoring tool [50]. They find that 90% of the refactoring edits are done manually. Kim et al. find that 51% of developers always preform manual refactorings without refactoring tools [37]. Negara et al. find that most expert developers perform refactorings manually instead of applying them by using refactoring tools [53]. Other studies also observe refactoring tools underused due to usability problems [51], unawareness [77], and insufficient trust (or distrust) [12, 70, 71, 77].

Manual refactoring can be tedious and error-prone as it often requires complex and multiple edits across different locations of a software system. According to Dig et al.’s study, over 80% of the API breaking changes in existing applications are refactorings [13]. Weißgerber and Diehl find a correlation between refactoring edits and increased bug numbers that are reported [79]. Kim et al. find that an increasing number of bug fixes is followed by API-level refactorings [35].

Several approaches are proposed to avoid refactoring anomalies. Formal verification can provide constraints to prevent refactoring edits from breaking behavioral changes of refactorings [48, 11, 57]. However, these approaches focus on improving the correctness of automated refactoring through formal specification, as opposed to finding anomalies during manual refactorings. Similar to refactoring tools in IDEs adopting condition checking, Ge and Murphy-Hill present refactoring condition checkers to ensure the correctness of refactorings and detection of defects introduced by manual refactorings [19]. In contrast to their approach to leveraging Eclipse refactoring APIs, we apply regression testing to re-run existing tests for validating the correctness of the refactored version of a program.
2.5 Change Impact Analysis

*Change impact analysis* [9, 63, 62, 67] aims to determine the semantic impact of a set of source code changes on other parts of a program. Ren et al. [63, 62] propose Chianti, which selects a subset of regression tests whose behavior might have changed and then identifies affecting changes responsible for test failure. For each test failure, Chianti could produce the number of affecting changes, some of which are irrelevant to refactoring changes that a developer need to inspect. Also, it could identify failed tests that interleave multiple asserts that can be partitioned into a set of test-slices, each test-slice is more semantically cohesive to alleviate many challenges for program understanding and debugging.

2.6 Fault Localization

Debugging is a labor-intensive task in maintenance activities. To accelerate this task, several approaches, including selective regression testing [1, 5, 24, 31, 80], have been proposed to improve fault localization, aiming at reducing tests that must be rerun after a software change. These approaches rank and reduce the set of potential faulty statements based on *program spectra*—an execution profile indicating which parts of a program are passed and failed during executions of a program. Although they reduce the cost of running regression tests, these approaches are not applicable to large evolving software systems, since they compute the execution profile on all statements in each program version. However, we leverage information about refactoring edits between old and new versions, and assist developers with understanding the impact of refactoring changes.

Zeller introduces the *delta debugging* approach [81] that identifies change subsets to generate intermediate program versions and localizes failure-inducing changes among large change sets. The key differences with our work are that our structural constraints before and after applying a refactoring to a program are analyzed to identify change subsets, whereas Zeller considers all changes between old and new program versions as a candidate set. Furthermore, Zeller determines a set of edits that may cause a test failure but does not decompose a test into statement subsets that should be inspected together for each failed assert, leaving it to a developer to examine a real culprit of a regression test failure among a likely large regression test suite and potential failure-inducing changes.
CHAPTER 3

PATTERN-BASED CLONE REFACTORING INSPECTION

This chapter starts with the following section that motivates our research and describes PRI with a real example drawn from the JEdit (http://jedit.org) project, which is an open source project—a text editor written in Java—with 120K lines of source code over 580 Java source files.

3.1 Motivating Example

Suppose Alice changes JEdit’s source code after she has encountered duplicated regions. She manually applies Extract Method to two cloned regions in methods processKeyEvent and processKeyEventV2\(^1\), respectively. She validates manual refactorings [77] using existing test cases; however, she misses refactoring one clone instance of the group in a different method processMouseEvent in Figure 3.1(c). For another clone instance in processActionEvent in Figure 3.1(d), she has difficulty conducting the same refactoring due to the type variation of variable changeEventAction, which is different from corresponding variables of other clone siblings in the same group.

To confirm that there is no location Alice missed to refactor during peer code review, Barry needs to investigate line level differences file by file. When Barry finds suspicious locations, he might want to inspect differences between Alice’s and subsequent revisions. Simply comparing with the newest revision can require Barry to decompose countless irrelevant changes [4, 73]. Since understanding such composite changes require non-trivial efforts [73], sub-changes that are aligned with Alice’s refactoring changes must be investigated by manually comparing Alice’s changes with other changes committed in subsequent revisions.

The following shows how Barry may use PRI to inspect Alice’s changes. First, he checks out revision \(r_i\) that she commits changes, and then he runs PRI, a plug-in built

\(^1\)Method processKeyEventV2 is created to support the different version.
public void processKeyEvent(KeyEvent evt, int from) {
    Event focusKeyTyped, focusKeyPressed;
    switch(evt.getID()) {
        case Event.KEY_TYPED:
            if (inputHandler.isActive() && from != VK_CANCEL) {
                focusKeyTyped = evt.getEvent();
            }
            focusKeyTyped = processEvent(from, focusKeyTyped);
        case Event.KEY_PRESSED:
            if (inputHandler.isActive() && from != VK_CANCEL) {
                focusKeyPressed = evt.getEvent();
            }
            focusKeyPressed = processEvent(from, focusKeyPressed);
    }
    Event processEvent(int from, Event event) {
        if (inputHandler.isActive() && from != VK_CANCEL) {
            focusKeyTyped = evt.getEvent();
        }
        return event;
    }
}

(a) A clone refactoring in revisions v20060919-7074 and v20060919-7075 in JEdit.

public void processKeyEventV2(KeyEvent evt, int from, int enhancedVer) {
    Event focusKeyTypedEvt, focusKeyPressedEvt;
    switch(evt.getID()) {
        case Event.KEY_TYPED:
            if (inputHandler.isActive() && from != VK_CANCEL) {
                focusKeyTypedEvt = evt.getEvent();
            }
            focusKeyTypedEvt = processEvent(from, focusKeyTypedEvt);
        case Event.KEY_PRESSED:
            if (inputHandler.isActive() && from != VK_CANCEL) {
                focusKeyPressedEvt = evt.getEvent();
            }
            focusKeyPressedEvt = processEvent(from, focusKeyPressedEvt);
    }
    Event processEvent(int from, Event event) {
        if (inputHandler.isActive() && from != VK_CANCEL) {
            focusKeyTyped = evt.getEvent();
        }
        return event;
    }
}

(b) A refactoring to clones of the same group in a later revision by other developer.

void processMouseEvent(MouseEvent evt, int src) {
    Event focusMousePressed;
    switch(evt.getID()) {
        if (inputHandler.isActive() && src != VK_CANCEL) {
            focusMousePressed = evt.getEvent();
        }
    }
}

(c) A clone instance that Alice misses to apply a same refactoring as (a) and (b).

void processActionEvent(ActionEvent evt, int where) {
    Action changeEventAction;
    switch(evt.getID()) {
        if (inputHandler.isActive() && where != VK_CANCEL) {
            changeEventAction = evt.getEvent();
            // The type of changeEventAction differs from aligned variables in other siblings in the clone group.
        }
    }
}

(d) A clone instance in the clone group, which is not refactorable unlike (a) ~ (c).

Figure 3.1: A simplified example: A clone group is refactored inconsistently. Cloned regions are highlighted, deleted code is marked with ‘-’ and added code marked with ‘+’.

atop Eclipse IDE. Based on $r_i$, PRI tracks $r_{i}, r_{i+1}, ..., r_{i+n}$, and summarizes clone refactoring performed by Alice (Figure 3.1(a)) and applied by other developers reusing the existing code by altering copy-pasted code (Figure 3.1(b)). The results include refactored revisions, types (e.g., Extract Method), locations (e.g., package, class, method, and line number) and restructuring descriptions (e.g., “Method m1 is refactored and cloned...
Figure 3.2: A viewer in PRI that shows clone refactoring summarization and refactoring anomaly detection regarding Figure 3.1.

code fragments are replaced with a call to an extracted method \( m2 \)). PRI also detects unrefactored clones in a clone group (Figure 3.1(c)), classifying if a clone instance is locally refactorable by standard refactorings [17] (Figure 3.1(d)).

Figure 3.2 shows a snapshot of a PRI’s viewer, which shows a clone group that comprises six clone instances searched in each row of a tree viewer. The first two rows show the clone instances factored out by Extract Method in revision 7075, and the next two rows show the clone instances refactored in the same way in the next revision 7076. PRI marks the fifth clone instance with symbol \( \times \), meaning that it detects an omission error (Figure 3.1(c)). It marks the last clone instance with symbol \( \times \), meaning that it discovers Alice’s difficulty in applying Extract Method in the same way due to the type variation (Figure 3.1(d)).\(^2\) The viewer in Figure 3.2 is synchronized with a visualization view designed for inspecting clone refactorings regarding structural and dependence relationships between clone instances and related contexts.

The aforementioned incomplete refactorings do not produce compilation errors, passing all existing test cases. Reviewers are likely to overlook these locations. In fact, it is not always possible to factor out all clones but independent evolution might be required in some clone instances; however, revealing these locations and classifying them whether to easily be removed using standard refactoring techniques can be worthwhile during a code review.

3.2 Approach

This section presents PRI consisting of the following three phases. Phase I summarizes clone refactorings using templates, which check the structural constraints before

\(^2\)Java generic types could be applicable for a refactoring, which is not included in standard refactorings [17].
and after applying a refactoring to a program and encode ordering relationship between refactoring types. Phase II detects clone groups incompletely refactored, classifying the reasons. Phase III provides code visualization to represent historical refactoring edits that Phases I finds and refactoring opportunities that Phases II detects. Figure 3.3 outlines the workflow of PRI as a pipeline that successively summarizes clone refactorings and helps to remove refactorable clones in the incomplete refactorings.

Figure 3.3: Overview of PRI’s workflow.

3.2.1 Phase I: Change Summarization for Clone Refactorings

Converting Clone to AST Model. Our approach parses clone groups reported by Deckard a clone detector [29]. PRI finds the set of AST nodes that contains reported clones:

\[
\{ n \mid \text{offset}(\text{clone}) \geq \text{offset}(\text{statements}) \land \\
\text{length}(\text{clone}) \leq \text{length}(\text{statements}) \land \\
n \in \text{ast}(\text{statements}) \}\}
\]

where offset(clone) is the starting position of clone instance clone from the beginning of the source file, and length(clone) is the length of the clone instance. The function ast(statements) finds an AST at the method level, and analyzes the AST to find inner-most syntactic statements (i.e., least common ancestor) that may contain incomplete syntactic regions of cloned code fragments. PRI improves the performance of search by caching ASTs of syntactic clones after computing Equation 3.1. The tool for parsing Java source code and generating the corresponding ASTs is provided with the Eclipse JDT framework.4

Tracking Clone Histories. PRI accesses each subsequent revision \( r_j \in R = \{ r_{i+1}, \ldots, r_n \} \), and identifies ASTs that are related to clone instances in an original revision \( r_i \).

3The default settings with 30 minT (minimal number of tokens required for clones), 2 stride (size of the sliding window), and 0.95 Similarity are used.

4http://www.eclipse.org/jdt/
PRI presents a *clone refactoring aware approach* to checking clone change synchronization across revisions. It is integrated with software configuration management (SCM) tools using an SCM library\(^5\) to analyze refactorings of individual clones and groups, helping developers focus their attention on any revision.

Returning to the motivating example, CI\(_1\) in `processKeyEvent` is a clone of a fragment CI\(_2\) in `processKeyEventV2` in the clone group. At revision \(r_1\), changes of `Extract Method` to CI\(_1\) are checked in, and changes of the same refactoring to CI\(_2\) are committed to the next revision \(r_2\). To track the evolved clone group, PRI automatically accesses the consecutive revisions \((n \geq 2, \text{where } n \text{ is configurable})\) to search for the clone siblings (CI\(_1\) and CI\(_2\)) by comparing two revisions.

**Extracting Changes between Two Versions of ASTs.** PRI leverages ChangeDistiller [15] to compute AST edits to code clones. We chose ChangeDistiller since it represents concise edit operations between pairs of ASTs by identifying change types such as adding or deleting a method, inserting or removing a statement, or changing a method signature. It also provides fine-grained expression with a single statement edit. PRI uses ChangeDistiller to compute differences between the clone ASTs in revision \(r_i\) and the evolved clone ASTs in revision \(r_j\).

Continuing with our motivating example, PRI parses the clone region before changes as shown in Figure 3.4(a), and parses the corresponding region after changes as shown in Figure 3.4(b). It then uses ChangeDistiller to compute the differences between two sets of ASTs. In Figure 3.1(a), ChangeDistiller reports the deletion operations of the statements including `if`, `then`, and `else`, and the insertion operations of an assignment statement including a call to a new method `processEvent`.

\(^5\)http://www.svnkit.com/
When checking changes before and after clone refactorings, it is important to determine if references (e.g., method call and field access) are preserved across clone instances. Therefore, we create a new reference binding checker, which is not provided by ChangeDistiller, to assess reference consistency. For example, Figure 3.4(b) shows a method invocation dependency between a caller in node \( N_9 \) and a callee in \( N_{10} \). We check if this reference association is preserved in other regions after changing clone instances by using bindings to a method.

Matching Clone Refactoring Pattern Templates. Refactoring pattern templates are AST-based implementations that consist of a pair of pre- and post-edit matchers such as \( M_{\text{pre}} \) and \( M_{\text{post}} \). \( M_{\text{pre}} \) is an implementation for matching patterns before clone refactoring application. \( M_{\text{post}} \) interacts with repositories and traverses ASTs of the source code in which the clones and their dependent contexts are modified. It extracts both a match between the nodes of the compared AST subtrees before and after a refactoring application and an edit script of tree edit operations transforming an original into a changed tree.

After matching such clone refactoring patterns comprising a set of constraint descriptions where a refactoring can be performed, \( \text{PRI} \) identifies concrete clone refactoring changes (e.g., refactored revisions, refactoring types, locations, and restructuring descriptions). The change pattern descriptions are designed by using declarative rule-based constraints [60].

Table 3.1 shows clone refactoring templates that our approach can identify based on pre- and post-edit matchers. We leverage the rules of refactoring types in Fowler’s catalog [17]. A composite refactoring comprises a set of low-level refactorings. For example, template 6 describes that Extract Method is applied to a clone group, and the new method is moved to another class.

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EM</td>
<td>Extract Method Refactoring</td>
</tr>
<tr>
<td>2</td>
<td>MM</td>
<td>Move Method Refactoring</td>
</tr>
<tr>
<td>3</td>
<td>PM</td>
<td>Pull Up Method Refactoring</td>
</tr>
<tr>
<td>4</td>
<td>ES</td>
<td>Extract Superclass Refactoring</td>
</tr>
<tr>
<td>5</td>
<td>MN</td>
<td>Move Type to New File Refactoring</td>
</tr>
<tr>
<td>6</td>
<td>EM+MM</td>
<td>Extract and Move Method Refactoring</td>
</tr>
<tr>
<td>7</td>
<td>EM+PM</td>
<td>Extract and Pull Up Method Refactoring</td>
</tr>
<tr>
<td>8</td>
<td>PM+EM</td>
<td>Pull Up and Extract Method Refactoring</td>
</tr>
</tbody>
</table>

Table 3.1: Clone refactoring templates that \( \text{PRI} \) supports by tracking the evolution of clones focusing on their removal refactorings.

Algorithm 1 illustrates our approach following this clone refactoring pattern identification. Our approach first takes a revision scope (REVs) for tracking clone refa-
Algorithm 1: Identifying Clone Refactoring Evolution.

**Input**: REVs – a scope of revisions for inspection, OCD – the output of a clone detector, and PRG – a program to be inspected.

**Output**: RES – a set of clone groups whose clone instances are classified as refactored, refactorable, or unfactorable.

1. **Algorithm main**
   2. $\text{CGs} \leftarrow M_{\text{pre.match}}(\text{OCD, PRG})$
   3. foreach $\text{Revision } r_i \in \text{REVs}$ do
   4.   foreach $\text{CloneGroup } G_{\text{clone}} \in \text{CGs}$ do
   5.     $\text{RES} \leftarrow M_{\text{post.match}}(G_{\text{clone}}, r_i)$
   6.   end
   7. end

Tracking histories and the output of a clone detector (OCD) as input. To match pre-edit patterns with OCD, it traverses the ASTs of OCD and extracts program elements (e.g., packages, classes, methods, and fields) and structural dependencies (e.g., containment, subtyping, overriding, and method calls). $M_{\text{pre.match}}$, a syntactic sugar, uses these predicates to determine clone structural patterns, such as whether they exist in the same structural location, or whether they are implemented in classes with the common superclass (Line 2). It then iterates two tasks: tracking clone groups across revisions (Line 3) and inspecting each clone group (Line 4). This iteration stops when all changed files in input revisions are inspected with all clone groups. Algorithm 1 returns the following results: (1) clone groups where all clone instances are refactored completely, (2) clone groups where no clone instance is refactored, and (3) clone groups where some of the clone instances are refactored. $M_{\text{post.match}}$ is a syntactic sugar for the invocation of any template to match with changes to clones (Line 5).

Continuing with our example, PRI matches patterns with changes to clones (Figures 3.1(a) and (b)) in each revision using templates. During this inspection, Algorithm 2 finds the pattern of Extract Method by performing rules in lines 3-4.

The following structural change matching rules capture Extract Method. (See more refactoring pattern templates at https://goo.gl/eFpG Gn)

- **Pattern 1**: deleteClone($ci_1$, $m_1$, $t_1$, $r_1$) – clone instance $ci_1$ is deleted from method $m_1$ in type $t_1$ in revision $r_1$.
- **Pattern 2**: addMethod($m_2$, $t_1$, $r_1$) – method $m_2$ is added in type $t_1$ in revision $r_1$.
- **Pattern 3**: addCall($m_1$, $m_2$, $r_1$) – a method call to $m_2$ from method $m_1$ is added in revision $r_1$. 
**Algorithm 2:** Matching the Extract Method Refactoring Pattern.

```
Input : G – a clone group, R – a revision, and Sim – a similarity threshold.
Output: RES – clone instances identified as refactored or not.

Template extractMethodRefPattern
foreach CloneInstance ci ∈ G do
  if deleteClone (ci, m_i, t_i, R) ∧ addMethod (m_j, t_i, R) ∧
    addCall (m_i, m_j, R) ∧ similarBody (m_i, m_j, Sim) then
    RES ← summarize (ci); // clone refactoring
  end
  else
    RES ← detect (ci); // refactoring anomaly
  end
end
```

- **Pattern 4:** similarBody(ci_1, m_1, m_2, Sim) – the similarity level between a deleted ci_1 in m_1 and a method body of m_2 is greater than threshold Sim.

We believe our approach can be easily extended to support other clone refactorings, which may reuse similar constituent change steps.

Our clone tracking technique for pattern similarBody uses the *Levenshtein distance* algorithm [42], which measures the similarity between two sequences of characters based on number of deletions, insertions, or substitutions required to transform one sequence to the other.

Our approach maps a code snippet M_i in the clone group and the edited statements M_j related to changes to M_i (e.g., clone instances and the body of method processEvent in Figure 3.1(a)). When comparing M_i and M_j, our approach generalizes the names of identifiers with abstract variables (e.g., variables, qualifier values, fields, and method parameters) to create a generalized program comparison that does not depend on concrete identifiers. This generalization technique was used in previous works [83, 47], and achieved a high performance to distinguish if two code fragments are relevant. We define the similarity S between M_i and M_j as follows:

\[
S = 1 - \frac{\text{LevenshteinDistance}(M_i, M_j)}{\max(|M_i|, |M_j|)}
\]  

(3.2)

The value of S is in the interval (0, 1]. Our technique in similarBody searches for a portion of code fragments from method m_2 with a similarity S of at least a threshold Sim when compared to a clone instance ci_1 deleted in method m_1.
One or more clone instances (CIs) are omitted to apply refactorings in the clone group; however, other clone siblings are refactored.

CIs use different variable types compared to types of aligned AST nodes in other clone siblings. To handle variations in types, a *parameterize type* refactoring can be considered [47]. The applicability of this refactoring is affected by language support for generic types.

CIs use different method calls compared to method calls of aligned AST nodes in other clone siblings. To handle variation in method calls, *Form Template Method* (pg. 345 in [17]) could be applicable by creating common APIs in the base class and encapsulating the variation in the derived classes.

CIs use different references (e.g., method overriding) compared to references of aligned AST nodes in other clone siblings. Similar to *dynamic-dispatch rule* [10], altering inheritance relations is checked if addition or deletion of an overriding method may result in reference changes.

CIs are implemented in different orders compared to execution orders in other clone siblings (e.g., n1(); n2(); vs. n2(); n1();). *Form Template Method* could be used to allow polymorphism to ensure the different sequences of statements proceed differently.

Non-syntactic CIs are detected, and syntactic clones expanding clone regions are not similar between clone siblings according to a threshold (*Sim*). A possible refactoring type is *Form Template Method* by implementing the details of the different process in the derived classes depending on semantic constraints or the length of common subsequent code.

Table 3.2: The six types of classification that PRI annotates classified clone instances with symbols in the first column S.

### 3.2.2 Phase II: Detecting Incomplete Clone Refactorings

PRI detects incomplete clone refactorings—there are clone instances which are unrefactored inconsistently with other sibling in the group. It also classifies unrefactored clones if they are locally refactorable or not. Non-locally-refactorable clones means that a developer has difficulty performing refactorings to remove clones using standard refactoring techniques [17] due to limitations of a programming language or incomplete syntactic units of clones.

To detect unrefactored clone instances, we reuse Equation 3.2. Our approach extracts differences of changes to clones, generalizes statements before and after changes, and computes a distance between \( M_i \) and \( M_j \). If the value of \( S \) is less than \( Sim \), our approach considers \( M_i \) as the unrefactored one and attempts to discover a possible reason why developers have not removed \( M_i \) by a refactoring.

Table 3.2 summarizes six types of unrefactored clones which can be automatically classified by PRI. We categorize these cases by manually investigating clone groups from six subject applications used in our case studies and plan to add more cases in the future.

Continuing with our example, PRI analyzes an alignment between a pair of two different strings “focusKeyTyped” and “changeEventAction” in Figures 3.1(a) and (d). It then maps their locations to AST nodes and searches their declarations by perform-
ing static interprocedural slicing [27] to determine if their types share a common type. PRI annotates with symbol $\mathbf{x}$ a clone instance in Figure 3.1(d), which is not factored out unlike other clone siblings due to the type variation. Figure 3.5 shows how PRI reports detection results to help developers inspect clones for refactorings. A$_1$, A$_2$, B$_1$ and B$_2$ in Figures 3.1(a) and (b) are summarized as refactorings; however, C and D in Figures 3.1(c) and (d) are detected as anomalies.

Figure 3.5: Detecting refactoring anomalies in clone refactoring evolution.

Figure 3.6: Visualizing clone refactorings and anomalies regarding Extract Method (see more screenshots at http://faculty.ist.unomaha.edu/msong/pri/).

---

6Generic types could be considered to remove the clone, but Fowler’s catalog does not include such techniques and current refactoring engines, such as Eclipse, do not support it.
3.2.3 Phase III: Visualizing Clone Refactorings and Anomalies

As PRI is intended for interactive use, we have implemented it in the context of the Java editor of Eclipse, a widely used extensible open-source development environment for Java. PRI shows the clone refactoring visualization view that graphically represents refactoring edits and anomalies in a clone group in a tree graph view. Continuing with our example, Figure 3.6 shows a snapshot of this view. We represent the structural relationship in a clone group with a solid line denoting their locations (i.e., line number); the class View (blue) contains six clone instances. We also represent the reference dependence with a dotted line indicating refactoring histories (i.e., revision); the four clone instances (green) refactored in processKeyEvent and processKeyEventV2 are linked to method updateFocus (yellow). The two unrefactored methods (red) are marked as refactoring anomalies without a link to the extracted method updateFocus.
CHAPTER 4

REFACTORINGS INVESTIGATION AND TESTING TECHNIQUE

This chapter describes the proposed work, starting with the below section as our motivation that illustrates how Rit would help a developer in inspecting and testing changes by standard refactorings [17].

4.1 Motivating Example

Suppose Sara works as a developer in a program in Figure 4.1. Sara notices an opportunity to remove code duplication (i.e., code clone removal) in the original version in Figure 4.1(a). The first and second clone groups\(^1\) comprise methods \(m1()\) and \(m3(E)\) in classes \(B\) and \(C\), respectively. The last clone group consists of clone instances in methods \(m5()\) and \(m6()\) in class \(B\). She decides to perform refactorings manually—Pull Up Method to the first and second clone groups respectively and Extract Method to the last clone group. Figure 4.1(b) shows the changes made by Sara, whose code insertion is marked with ‘+’ and deletion with ‘-’.

Sara performs two Pull Up Method and one Extract Method refactorings in Figure 4.1(b): (1) the first Pull Up Method moving methods \(m1()\) from class \(B\) and \(C\) to class \(A\), (2) the second Pull Up Method similar to the first one moving from methods \(B.m3(E)\) and \(C.m3(E)\) to method \(A.m3(E)\), and (3) Extract Method creating method \(B.m4()\) by extracting common code fragments from methods \(B.m5()\) and \(C.m6()\).

While the first Pull Up Method does not break behavior preservation, Sara does not notice that the second Pull Up Method forms method overloading by creating \(A.m3(E)\), which has the same method name as \(A.m3(F)\) with different implementation. Sara does not suspect that she mistakenly changed the program semantics. She can find failures by running entire regression test suites; however, running all of the test cases in a test suite can require a large amount of effort to ensure the refactoring correctness.

\(^1\)A clone group is a set of two or more clone instances—the similar code fragment.
When using RIT for inspecting and validating her refactoring, Sara or other developers would easily detect the refactoring anomaly since RIT applies a static analysis to identify refactoring changes. It then selects a subset of regression tests whose behavior might have been changed and then reports affecting refactoring—Pull Up Method to
```java
@Test
public void test1() {
    A b1 = new B();
    A c1 = new C();
    b1.m1();
    c1.m1();
    Lib.println(b1.g + " eq " + c1.g);
    assertTrue(b1.g == c1.g);
}

@Test
public void test2() {
    A b2 = new B();
    int rt1 = 0, rt2 = 0;
    rt2 = field1;
    if (field1 != -1)
        rt1 = b2.m4();
    A a = new A();
    E e = new E();
    e.foo(a);
    rt2 = a.m3(e);
    Lib.println(b2.g + " eq " + rt1);
    Lib.println(a.k + " < " + rt2);
    assertTrue(b2.g == rt1);
    assertTrue(a.mk() < rt2);
}
```

Figure 4.2: Regression tests `test1()` and `test2()` for testing an example program in Figure 4.1.

B.m3(E) and C.m3(E) for creating A.m3(E) responsible for a test failure in `test2()` Figure 4.2.

Sara performs Extract Method in Figure 4.1(b). She removes code clones from methods B.m5() and B.m6() to create a common method B.m4(). However, she unexpectedly introduces refactoring anomaly by overriding method A.m4() with the new method B.m4(), leading to changes to semantic preservation. After running all regression test suites, she finds test failure. It may not be cost-effective when faults should be quickly detected. Furthermore, she might not be easy to understand a composite test that interleaves multiple asserts and to debug the root cause(s).

RIT finds composite tests interleaving two or more asserts (e.g., test `test2()` in Figure 4.2). Given the assert, RIT applies program slicing [78] to identify the test context—preceding code relevant to variables for tested data in the assert in terms of data dependences (e.g., a variable definition and its use at lines 23, 24, 27, 34, and 37 and at lines 24, 29, 30, 31, 32, 35, and 38 in `test2()` in Figure 4.2)\(^2\) and control dependences (e.g., the if statement from lines 26 to 27), which are likely to be related and should be inspected and tested together. RIT forms partitions that can be understood and tested independently, and produces each reduced, executable test-slice including relevant statements and control predicates that directly or indirectly affect the variables used in asserts.

Given a set of partitioned test-slices, RIT further minimizes a test-slice by forward/backward tracking tested variables referenced in asserts. For example, it excludes read statements referencing a tested variable without affecting its value, since such read statements are unlikely to affect test behaviors (e.g., statements at lines 8, 34 and 35 in Figure 4.2).\(^3\) However, it determines write statements altering the value

\(^2\)An assignment at line 25 is excluded since it is killed (redefined) by a statement at line 32.

\(^3\)Unrelated asserts are also excluded exclusively. For example, a test-slice for assert at line 37 excludes `assert` at line 38 as a read statement and vice versa.
public void test2_1() {
    A b2 = new B();
    int rt1 = 0;
    if (field1 != -1)
        rt1 = b2.m4();
    assertTrue(b2.g == rt1);
}

(a) A partitioned test-slice test2_1() which is separated from a composite test test2() in Figure 4.2. This test is affected by the Extract Method refactoring.

public void test2_2() {
    A a = new A();
    E e = new E();
    e.foo(a);
    int rt2 = a.m3(e);
    assertTrue(a.mk() < rt2);
}

(b) A partitioned test-slice test2_2() from test2() separated from a composite test test2() in Figure 4.2. It is affected by the Pull Up Method refactoring.

Figure 4.3: Generating subtests each of which address an independent development and maintenance concern.

define a tested variable responsible for test behaviors (e.g., variable rt2 updated by an assignment statement at line 32 in Figure 4.2). Regarding a call to E.foo(A) at line 31, it finds aliases of the tested variable whose heap object gets modified by tracking the respective object (e.g., x.k at line 66 in Figure 4.1(a)). To track tested variables and search for aliases for the target variable, \( \text{RIT} \) applies both dynamic and static techniques for inter-procedural data flow analysis [64].

Figure 4.3 shows a set of test-slices that helps a developer focus on potential failure-inducing edits (i.e., affecting refactorings). \( \text{RIT} \) identifies affecting Pull Up Method by using the affected, partitioned test-slice test2_1() in Figure 4.3(a) and affecting Extract Method through the test-slice test2_2() in Figure 4.3(b). For example, given a test-slice test2_1() reporting a regression test failure, \( \text{RIT} \) identifies the affecting Pull Up Method to deletion of methods B.m3() and B.m3() with creation of method A.m3() in Figure 4.1(b). Given a test-slice test2_2() reporting a failure, \( \text{RIT} \) detects the affecting Extract Method to methods B.m5() and B.m6() with creation of method B.m4() in Figure 4.1(b).

Although such semantic behavior changes may be Sara’s intended edits during refactorings, it may be worthwhile for others to note these behavior changes or for Sara to reconfirm her intent, since these refactorings altered the external behavior of the program.

4.2 Approach

Given the original and edited program versions, \( P \) and \( P’ \), \( \text{RIT} \) first extracts differences between \( P \) and \( P’ \) (Section 4.2.1), analyzes dependences between changes (Section 4.2.2), and identifies refactorings. Second, it selects a subset of regression tests affected tests whose behaviors could be changed by the refactorings. We analyze a call graph to safely select affected tests that contain at least every test whose behavior may...
have been affected (Section 4.2.3).

Third, if a composite test containing multiple asserts, \textit{RIT} identifies semantically dependent statements to execute logically related statements regarding each assert. Related statements are then merged together and isolated from others. A composite test can be partitioned into a set of statement-slices (Section 4.2.4).

Forth, regarding a set of partitioned test-slices, \textit{RIT} minimizes each test-slice by removing irrelevant statements (i.e., read statements) to perform method assert. It determines statements modifying tested variables (i.e., write statements) within a test-slice by tracking these tested variables used in assert, applying data flow and aliases analysis techniques (Section 4.2.5).

Lastly, for each test-slice that causes failures, \textit{RIT} determines affecting changes (i.e., refactoring anomalies) that change behaviors of tests. Given a failed test, \textit{RIT safely} determines a set of affecting changes that contains at least every change that may have caused changes to the test’s behavior (Section 4.2.6). See the system overview in Figure 4.4.

4.2.1 Change Category

\textit{RIT} uses a differencing technique [15] to extract program edits as atomic changes by comparing ASTs of two program versions. These atomic changes are categorized as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), change method (CM), added fields (AF), deleted fields (DF), lookup (i.e., dynamic dispatch) change (LC), and resolution of overload (i.e., static dispatch) change

![Figure 4.4: The system overview of \textit{RIT}.](image-url)
(RC). Most of atomic changes are self-explanatory except for LC and RC. The look-up change LC can be caused by adding or deleting methods/fields, modifying inheritance relationships, and it is defined as a change rule, \( \text{LC} = <Y, X.m()> \), where \( Y \) denotes the run-time type of the receiver, and \( X.m() \) denotes the method that is statically referred to in the method call on an object with type \( X \). It models changes that the dynamic dispatch behavior for a call to method \( X.m() \) with run-time type \( Y \) leads to a different method. These change types were also used in previous works [63, 82].

RTI introduces the resolution of overload change (RC) that occurs as a result of method hiding changes due to a hierarchy of method argument types. For example, RC can be caused by adding multiple methods of the same name with different behaviors passing different argument types, while showing inheritance relationships between the argument types passed to the methods. It is defined as a change rule, \( \text{RC} = <Y', X.m(Y)> \), where \( Y' \) denotes the resolved type at compile-time, and \( X.m(Y) \) denotes the method that is statically referred to in the method call passing an argument
type $Y$, while types $Y$ and $Y'$ show an inheritance relationship.\textsuperscript{4} RC models changes that a call to method $X.m(Y)$ with type $Y'$ determined by the compile-time resolution result in the selection of a different method due to the inheritance relationship between types $Y$ and $Y'$.  

### 4.2.2 Change Dependences

\textbf{RIT} identifies dependences between atomic changes, leverages the structural constraints of a program before and after each refactoring type, and summarizes refactoring changes. It helps developers understand and debug refactored programs. Atomic changes in Figures 4.5 shows refactorings applied to the running example. An arrow from an atomic change $A_1$ to an atomic change $A_2$ indicates that $A_2$ is dependent on $A_1$ (i.e., $A_2$ is a prerequisite for $A_1$).

Consider, for example, the addition of the call $m4()$ in methods $m5()$ and $m6()$ during Extract Method. These changes occur in atomic changes C9 and C10 in Figure 4.5(b). Adding these calls breaks the semantic preservation without method $B.m4()$ during Extract Method according to Fowler’s catalogue [17]. Therefore, atomic changes C9 and C10 create dependence relationship with C12, which is an AM change of $B.m4()$ for extraction of common code fragments from methods $B.m5()$ and $B.m6()$.\textsuperscript{5}

For every LC change, there is method addition (AM) or deletion (DM) which caused it, and LC is dependent on these changes. Similarly, every LC change depends on field additions (AF) or deletion (DF). Consider, for example, Extract Method adding method $B.m4()$, changing methods $B.m5()$ and $B.m6()$ in Figure 4.1(b). As a result of this refactoring, a call to $A.m4()$ on an object of type B dispatches to $B.m4()$ in the refactored program, whereas it used to invoke $A.m4()$ in the original program in Figure 4.1(a).

Every RC change depends on corresponding AM or DM changes. Consider, for example, Pull Up Method adding method $A.m3(E)$, deleting methods $B.m3(E)$ and $C.m3(E)$ in Figure 4.1(b). This refactoring changes the behavior of the program because a call to $A.m3(F)$ passing an argument with type $E$ (a subtype of $F$) resolves to $A.m3(E)$ in the refactored program, whereas it used to invoke $A.m3(F)$ in the original program.\textsuperscript{6}

While previous techniques [63, 82] consider dependences among atomic changes,\textsuperscript{4} Methods are overloaded with different argument numbers; however, we emphasize that RC can be caused despite no edits in callers or callees.
\textsuperscript{5}There are two steps: the addition of an empty method $B.m4()$ (AM at C12 in Figure 4.5) and the insertion of the method $B.m4()$ (CM at C13).
\textsuperscript{6}The LC changes to $A.m3()$, $B.m3()$ and $C.m3()$ are skipped in Figure 4.5(c) due to limited space.
they do not model dependences between changes precisely when refactorings (along with behavior changes) are performed, which results in false positive dependences. Based on such rules, RIT models dependence relationships among atomic changes of refactorings more accurately.

4.2.3 Selecting Affected Tests

RIT constructs a dynamic call graph for each test by tracking the execution of the tests. Given a set of regression tests, it determines a subset of tests that is potentially affected by the changes in a set of atomic changes. It then correlates these changes against the call graphs for the tests in the original version of the program.

The call graph is defined as a representation, \( G = <V, E> \), where \( V = M \cup F \) and \( E = \bigcup_{m \in M} \{ m \to m_i | m_i \in \text{callee}(m) \} \cup \{ m \to f | f \in \text{access}(m) \} \). \( M \) and \( F \) denote a subset of methods and fields reachable from each test in the program under test, and \( \text{callee}(m) \) and \( \text{access}(m) \) denote a subset of called methods and accessed fields by \( m \) respectively.

Figure 4.6(a) shows the call graphs for two tests test1() and test2() before refactorings have been applied to the running example. A test is determined to be affected if its call graph in the original version of a program contains a node that correspond to CM or DM changes. Also, a test is selected if its call graph contains a node that correspond to LC or RC changes. Given the call graph in Figure 4.6(a), RIT determines that test1() and test2() are affected by refactorings, because test1() contains two edges corresponding to a dispatch to methods \( B.m1() \) on an object of type \( B \) and \( C.m1() \) on an object of type \( C \), which correspond to LC changes C3 and C4. The test test2() contains edges for LC and RC: (i) an edge corresponding to a dispatch to method \( A.m4() \) on an object of type \( B \), which corresponds to LC change C8, and (ii) an edge corresponding to a resolution to method \( A.m3(F) \) with the subclass argument type E, which corresponds to RC change C17.

4.2.4 Partitioning Affected Tests

RIT finds a composite test including multiple asserts and parses it into Abstract Syntax Tree (AST) edits, and it extracts the test context—preceding statements on which the tested variables in the assert are data and control dependent by performing static intraprocedural slicing [27]. It identifies all upstream dependent AST nodes based on a transitive relation within a test.
• Data dependence – Statement $S_2$ is data dependent on $S_1$, if $S_1$ defines a variable $v$ and $S_2$ uses $v$, such that there exists a path from $S_1$ to $S_2$ which $v$ is not killed (redefined).

• Control dependence – Statement $S_2$ is control dependent on $S_1$, if execution of $S_2$ is control dependent on the decision made at $S_1$.

**RIT** constructs a program dependence graph, which is defined as a representation, $PDG = <DN, DE>$, where $DN$ denotes a subset of statements, and $DE \subseteq DN \times DN$ representing the control and data dependences between statements. It then decompose a composite test into separate sub-tests (i.e., a set of test-slices) based on PDG as the primary organizing principle. Figure 4.6 shows the partitioned test-slices by applying program slicing to tests test1() and test2().

### 4.2.5 Minimizing Affected Tests

**RIT** generates a set of partitioned test-slices $S$ from each composite test for validating a set of tested variables $V_t$. It then separates read statements (i.e., statements with a reference to each tested variable $v_t$ of $V_t$ without an update to $v_t$ in the left hand side of an assignment) from each test-slice of $S$, which only includes write statements (i.e., statements modifying the value of $v_t$). This separation of irrelevant statements (i.e., read statements) that do not affect test-slices’ behaviors reduces the maintenance efforts for developers and helps them understand specific call dependencies where the changes to the source code are unlikely to be apparent in the program behavior [73, 26].

To distinguish such read statements from write statements, **RIT** analyzes the dependent test context by (1) tracking each tested variable $v_t$ of $V_t$ propagating across method calls and (2) using “shortcut rules” predefined in XML file format.

Firstly, **RIT** resolves aliasing by combining a forward-alias analysis and an PDG-based backward analysis [74]. Continuing with our running example, **RIT** deduces that each $v_t$ is modified at statements $S_3$ and $S_8$ in Figure 4.7 respectively. In step ①, based on a constructed PDG for a test-slice test2.2(), **RIT** tracks variables $a$ and $rt2$ in $V_t$ on dependent statements backward, starting from assert at a statement $S_{11}$. Regarding a statement at $S_8$, it considers it as a write statement, because variable $rt2$ as $v_t$ is updated in the left hand side of an assignment due to a call to $A.m3(E)$. Due to an update of the call to $v_t$ (i.e., $rt2$) at $S_8$, **RIT** preserves this statement to run the test-

---

7**RIT** analyzes AST nodes except the ones on control predicates in statements to search for read statements.
Figure 4.7: Applying data flow and aliases analyses to a test-slice `test2_2()` to analyze semantic dependencies between procedures using inter-procedural data flow information such as alias. $S_n$ denotes a sequence of statements on control flow graph nodes; a dotted-line a backward analysis direction; a solid line a forward analysis direction; a underlined statement a location identified by an aliases analysis, and; a dotted-circle a correspondence of propagated $v_t$. Despite an unperformed analysis within method `A.m3(F)` skipped by a preceding analysis result, the access of field variables `this.i` and `this.k` can be determined by our approach.

slice `test2_2()`, stopping steps 2 and 3 for a forward analysis to track the variable `a` as $v_t$.

As step 4 continues the tracking backward for `a` as $v_t$, the address of variable `a` as $v_t$ at $S_3$ is propagated across methods through CFG yielding the address value for a parameter `z` at method `E.foo(A)` in step 5. The forward analysis searches for aliases of the respective object (`z` in this case). At $S_4$, the alias `x` of the respective object `z` is found and then propagated forward. Step 6 continues the tracking for the alias. RIT then identifies an update of variable `w` to the alias `x.k` at $S_7$, and thus it preserves the statement at $S_3$ within the test-slice `test2_2()` after step 7.

RIT tracks backward and preserves statements $S_1$ and $S_2$ in PDG of the test-slice `test2_2()`, because $S_1$ is a definition of `a` in `V_t`, and $S_2$ is a definition of `e` that is used in write statements such as $S_3$ and $S_8$.

Secondly, RIT excludes read statements from a test-slice by matching system API calls, whose arguments corresponding to $v_t$ of method calls are not updated (e.g., argument(s) of a call to `System.out.println`). In Figure 4.7, RIT excludes a statement at $S_{11}$ marked with X. External libraries including the JRE platform runtime in the

\[\text{In this paper, RIT’s goal is to analyze subject applications written in Java.}\]
analysis potentially requires a large amount of analysis time due to approximations performed during the library’s analysis, resulting in undesired imprecision. Based on a set of predefined rules, RIT utilizes an interface for external library models. These rules handles a collection of classes and data structures that are commonly used. Heuristics to consider a data reachability algorithm in the call graph for libraries are planned as future work [84].

Continuing with our running example, RIT takes as input tests test1() and test2() with each PDG, partitions the composite test test2() into test-slices test2_1() and test2_2 including other tests like test1(), and minimizes these test-slices by excluding (i.e., read statements) in Figure 4.3.

RIT constructs PDG and analyzes a program data flow by using Eclipse JDT, a static analysis framework to track tested variables forward/backward. To build the call graphs, we utilize an aspect-oriented programming (AOP) technique [32, 33] for instrumenting the class files of the original version of the program and their tests. For the reuse of analysis results, call graphs are stored as XML files. Executing each test case that has been instrumented by the AOP tool produces an XML file containing the program’s dynamic call graph.

4.2.6 Selecting Affecting Refactoring Changes

RIT takes as input each test-slice s of a set of partitioned test-slices S. To identify the refactoring changes that affect a subset of affected s, it constructs a call graph for those affected s in the refactored version P’ of the program in Figure 4.6(b). For affected s, RIT selects the set of atomic changes: added methods (AM) and changed methods (CM) that correspond to a node in the call graph when running on P’. It also selects atomic changes in the lookup changes (LC) and the resolution changes (RC) that correspond to an edge in the call graph in P’. It determines transitively prerequisite atomic changes; otherwise, applying change results in a syntactically invalid program or breaks rules for behavior preservation of transformation in the context of refactoring.

Continuing with our running example, given a failed test-slice test2_1(), RIT selects AM<\texttt{B.m4()}> because the call graph includes an edge LC<\texttt{B, A.m4()}>. It also reports C8 – C13 as a set of related atomic changes for Extract Method. Given a failed test-slice test2_2(), RIT selects AM<\texttt{A.m3(E)}> since the call graph includes an edge RC<\texttt{E, A.m3(E)}>, revealing a set of dependent atomic changes, C14 – C19 for Pull Up

9http://eclipse.org/jdt
Method.

Given each test-slice $s$ of a set of partitioned test-slices $S$, selecting affecting atomic changes $A$ in refactorings is defined as follows:

$$AffectingChanges(s, A) =$$

$$\{ a' | s \in S, a \in Nodes(P', s) \cap (CM \cup AM), a' \preceq^* a \} \cup$$

$$\{ a' | s \in S, a \equiv \langle B, X.m \rangle \in LC, B <^* A \preceq^* X,$$

$$n \rightarrow \langle B, A.m \rangle \in Edges(P', t),$$

$$\text{for some } n, A.m \in Nodes(P', s), a' \preceq^* a \} \cup$$

$$\{ a' | s \in S, a \equiv \langle G, Y.m(H) \rangle \in RC, G <^* H, A \preceq^* Y,$$

$$n \rightarrow \langle G, A.m(H) \rangle \in Edges(P', t),$$

$$\text{for some } n, A.m(H) \in Nodes(P', s), a' \preceq^* a \}$$

In the above definition, we express a transitive ordering using $\preceq^*$ to denote dependence relationships between atomic changes of refactorings.\(^{10}\) We represent a transitive hierarchy using $<^*$ and $\preceq^*$ to denote inheritance relationships between classes.\(^{11}\) We use $\rightarrow$ to denote call edges between methods.

\(^{10}\) For $\forall a' \in A$ such that $a' \preceq^* a, a \in A' \Rightarrow a' \in A'$. Given a set of $A$ of atomic changes that transforms $P$ into $P'$, applying $A'$ to $P$ results in a syntactically valid program with a guarantee of program behavior preservation for refactorings.

\(^{11}\) $A <^* B$ means $A$ is a subtype of $B$, but $A \neq B$, and; $A \preceq^* B$ means $A$ is a subtype of $B$, or $A = B$. 
CHAPTER 5

EVALUATION

To guide our investigation, we define the following research questions:

- **RQ1.** Can PRI accurately summarize clone refactorings?
- **RQ2.** Can PRI accurately detect incomplete clone refactorings?
- **RQ3:** Can RIT accurately determine test-slices affected by atomic changes of refactorings?
- **RQ4:** Can RIT accurately detect affecting refactorings that cause the failure of these test-slices?

5.1 Case Studies for Clone Refactoring Inspection

5.1.1 Experimental Design for RQ1 and RQ2

To evaluate PRI, we collect the data set by manually examining clone groups and their changes where real developers applied clone refactorings in repositories. Table 5.1 shows details of six subject applications used in our evaluation. We select these projects for two main reasons. First, all subject applications are written in Java, which is one of the most popular programming languages.\(^1\) Second, these applications are under active development and are based on a collaborative work with at least 48 months of active change history.

To measure PRI’s capability, we establish a ground truth set from six subject applications in Table 5.1 in the following steps. First, we parse commit logs to a bag-of-words and stemmed the bag-of-words using a standard NLP tool [46]. We then use keyword matching (e.g., “duplicated code” and “refactoring”) in the stemmed bag-of-words to find corresponding revisions. Based on these revisions, we manually investi-

\(^1\)http://www.tiobe.com/tiobe-index/
gate changed files to find clone refactorings that real developers performed or missed one or more clone instance in the same clone group.

To measure PRI’s capability to track the clone evolution, we manually decompose refactorings into individual changes and committed these changes across revisions to our evaluation repository.

Each clone refactoring is a pair of \((P, P')\) of a program, where \(P\) is an original version with clone groups, and \(P'\) is a new version that factors out clone groups. If refactorings in \(P'\) are performed across revisions by completely removing clone groups in \(P\), we add these changes in our data set \(G_1\) to evaluate RQ1. If any clone instance of a clone group in \(P\) remains unrefactored in \(P'\) within 10 subsequent revisions, we add these incomplete clone refactorings as anomalies in our data set \(G_2\) to evaluate RQ2.

Using the ground truth data set \(G_1\), precision \(P_1\) and recall \(R_1\) are calculated as

\[
P_1 = \frac{|G_1 \cap S|}{|S|}, \quad R_1 = \frac{|G_1 \cap S|}{|G_1|},
\]

where \(P_1\) is the percentage of our summarization results that are correct, \(R_1\) is the percentage of correct summarization that PRI reports, and \(S\) denotes the clone groups identified by PRI, all clone instances of which are refactored. Using the ground truth data set \(G_2\), precision \(P_2\) and recall \(R_2\) are calculated as

\[
P_2 = \frac{|G_2 \cap D|}{|D|}, \quad R_2 = \frac{|G_2 \cap D|}{|G_2|},
\]

where \(P_2\) is the percentage of our detection results that are correct, \(R_2\) is the percentage of correct detection results that PRI reports, and \(D\) indicates the clone groups detected by PRI, some clone instances of which are not refactored. We measure accuracy using F1 score by calculating a harmonic mean of precision and recall.

\[\text{As } 37\% \text{ of clone genealogies last an average of 9.6 revisions in Kim et al.'s study [36], we chose 10 subsequent revisions.}\]
Table 5.2: Accuracy of PRI’s summarization and detection. **RFT**: the refactoring types that developers perform across **VER** revisions (see Table 3.1 for acronyms), **VER**: the number of revisions where developers apply refactorings, **TIM**: the time that PRI completes each task (an average of time (sec.) per group), **CL**: the number of refactored clones correctly summarized by PRI (instance/group), **GT**: the number of the ground truth data set for clone refactoring summarization (instance/group), **P**: precision (%), **R**: recall (%), and **A**: accuracy (%), and each line represents the evaluation result for a project at a particular revision where developers apply clone refactoring applications.

<table>
<thead>
<tr>
<th>ID</th>
<th>RFT</th>
<th>VER</th>
<th>TIM</th>
<th>CLs</th>
<th>GTs</th>
<th>P</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EM</td>
<td>3</td>
<td>1.1</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>PU</td>
<td>3</td>
<td>0.1</td>
<td>5/2</td>
<td>6/3</td>
<td>100</td>
<td>66.7</td>
<td>80.0</td>
</tr>
<tr>
<td>3</td>
<td>PU</td>
<td>4</td>
<td>0.4</td>
<td>6/1</td>
<td>7/2</td>
<td>100</td>
<td>50.0</td>
<td>66.7</td>
</tr>
<tr>
<td>4</td>
<td>ES</td>
<td>9</td>
<td>0.9</td>
<td>20/4</td>
<td>20/4</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>ES</td>
<td>4</td>
<td>0.4</td>
<td>2/1</td>
<td>9/3</td>
<td>100</td>
<td>33.3</td>
<td>50.0</td>
</tr>
<tr>
<td>6</td>
<td>ES</td>
<td>3</td>
<td>0.8</td>
<td>17/7</td>
<td>17/7</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>ES</td>
<td>3</td>
<td>0.6</td>
<td>48/24</td>
<td>48/24</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>EM+PU</td>
<td>7</td>
<td>1.0</td>
<td>3/1</td>
<td>8/2</td>
<td>100</td>
<td>50.0</td>
<td>66.7</td>
</tr>
<tr>
<td>9</td>
<td>EM</td>
<td>3</td>
<td>0.8</td>
<td>4/2</td>
<td>4/2</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>EM</td>
<td>9</td>
<td>0.9</td>
<td>8/1</td>
<td>8/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>PU</td>
<td>4</td>
<td>0.7</td>
<td>9/3</td>
<td>9/3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>12</td>
<td>PU</td>
<td>3</td>
<td>0.5</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>13</td>
<td>PU</td>
<td>3</td>
<td>1.1</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>PU</td>
<td>3</td>
<td>0.3</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>15</td>
<td>PU</td>
<td>3</td>
<td>0.4</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>ES</td>
<td>3</td>
<td>2.3</td>
<td>40/20</td>
<td>40/20</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>17</td>
<td>PU+EM</td>
<td>3</td>
<td>2.2</td>
<td>6/3</td>
<td>6/3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>18</td>
<td>ES</td>
<td>3</td>
<td>1.4</td>
<td>11/5</td>
<td>12/6</td>
<td>71.4</td>
<td>83.3</td>
<td>76.9</td>
</tr>
<tr>
<td>19</td>
<td>EM+PU</td>
<td>3</td>
<td>1.5</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>EM</td>
<td>3</td>
<td>0.6</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>MN</td>
<td>3</td>
<td>2.4</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>22</td>
<td>PU</td>
<td>3</td>
<td>0.1</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>23</td>
<td>PU</td>
<td>3</td>
<td>0.5</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>24</td>
<td>EM</td>
<td>3</td>
<td>0.6</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>25</td>
<td>EM</td>
<td>3</td>
<td>0.6</td>
<td>2/1</td>
<td>2/1</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>26</td>
<td>ES</td>
<td>4</td>
<td>0.7</td>
<td>6/3</td>
<td>6/3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27</td>
<td>EM+MM</td>
<td>5</td>
<td>1.3</td>
<td>8/3</td>
<td>8/3</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

**TOTAL or AVG.** 103 0.9 217/92 232/98 98.9 92.0 94.1

5.1.2 Study Results and Discussion

Table 5.2 and Table 5.3 summarizes our evaluation result, to answer the questions raised above. We collect results for PRI and manually assess the outcomes to collect precision, recall, and accuracy. Regarding validation process, the first author analyzed PRI’s results. The results then were validated in the meetings with the remaining authors. When there was any disagreement, each issue was put to a second analysis round, and a joint decision was made.

**RQ1. Can PRI accurately summarize clone refactorings?** We assess PRI’s precision by examining how many of refactorings of clone groups are indeed true clone
Table 5.3: Accuracy of PRI’s summarization and detection. **RFT**: the refactoring types that developers perform across revisions, **CL**: the number of unrefactored clones correctly detected by PRI (instance/group), **GT**: the number of the ground truth data set for incomplete clone refactoring detection (instance/group), **X**, **Xt**, **Xm**, **Xo**, and **XS**: see Table 3.2, **P**: precision (%), **R**: recall (%), and **A**: accuracy (%), and each line represents the evaluation result for a project at a particular revision where developers apply clone refactoring applications.

PRI summarizes 94 refactorings of clone groups, 92 of which are correct, resulting in 98.9% precision. Regarding recall, PRI identifies 92% of all ground truth data sets. It identifies 92 out of 98 refactored groups. It summarizes refactorings of clone groups, tracking the clone histories with 94.1% accuracy.

PRI summarizes refactorings of clone groups that are not easy to identify because they require investigating refactorings of individual versions of a program while tracking changes of a clone group, e.g., a clone instance refactored in revision \( r_i \) and its sibling of the clone refactored in revision \( r_{i+j} \). Instead of tracking each version incrementally, simply comparing with the latest version can produce every change to be inspected. However, composite code changes, which intermingle multiple development issues together, are commonly difficult to conduct a code review [73].

**False Positives.** One group is a false positive due to the partitioning issue in clone...
For example, Extract Super Class is applied to a clone group in Apache Tomcat (r1742245). Although PRI identifies a refactored clone group, the group is not mapped to the ground truth as a clone detector detects the group as two separate groups. Selective merging analysis of clone instances of clone groups can prevent this false positive, which will be included in our heuristics in the future.

**False Negatives.** One group is not identified by PRI; the refactoring can only be identified by decreasing a similarity level in a clone detector. As PRI relies on the output of a clone detector, it is impossible to identify the refactoring of a group until a clone detector can find all sibling clones. In ArgoUML (r11784), we observed that changing the default setting from 0.95% to 0.85% can let PRI summarize refactorings of the group not reported when using the default setting.

Our threshold Sim also prevents PRI from identifying one group, since the reported cloned regions deviate from the regions that a refactoring is applied in real scenarios. For example, Extract Method is applied to smaller portions than the reported clones in ArgoUML (r16118). This false negative also negatively affects detection of incomplete clone refactorings with respect to precision since refactored clones, which PRI is unable to identify, is considered as unrefactored clones. Controlling Sim can allow to capture such issues.

Other groups not identified by PRI are modified after refactoring application in ArgoUML. We manually investigate the changes to the clone groups and find that refactoring edits includes extra edits without preserving the behavior after applying refactorings to clone groups. Checking the correctness of refactorings to determine whether extra edits are added to the pure refactoring version is our future work.

**RQ2. Can PRI accurately detect incomplete clone refactorings?** We estimate the precision of PRI by evaluating how many of the unrefactored clones are indeed a true omission of refactoring. As PRI detects clone groups in which some clone instances are omitted from refactorings either intentionally or unintentionally, we consider any instance resulting in refactoring deviations of other refactored siblings in the group as a true clone refactoring anomaly. PRI detects 1,162 unrefactored groups, 1,152 of which are correct, resulting in 99.4% precision. Regarding recall, PRI detects 97.8% of all ground truth data sets. It detects 1,152 out of 1,164 unrefactored groups with clone refactoring classification with 98.4% accuracy.

PRI helps developers investigate unrefactored clone instances and understand how these instances are diverged from other clone siblings. It automatically classifies whether unrefactored clone instances cannot be easily removed by standard refactoring techniques [17], which is not easy to determine since understanding clone differences
class AbstractAjpProtocol {
    try {...
        endpoint.pause();
        ... } catch {... }
    }

void pause() {
    try {...
        endpoint.pause();
        ... } catch {... }
    }

void resume(byte[] data) {
    try {...
        endpoint.resume();
        ... } catch {... }
    }

void stop(byte[] data) {
    try {...
        endpoint.stop();
        ... } catch {... }
    }
}

class AbstractHttp11Protocol {
    void pause() {
        try {...
            endpoint.pause();
            ... } catch {... }
        }
    }

void resume(byte[] data) {
    try {...
        endpoint.resume();
        ... } catch {... }
    }

void stop(byte[] data) {
    try {...
        endpoint.stop();
        ... } catch {... }
    }
}

Figure 5.1: A false negative example from the Apache Tomcat (r1042872) project (the regions with highlighted background are cloned).

between clone instances in the group usually requires both mapping aligned statements line by line and checking if these statements are implemented with the same program elements (e.g., types or method calls).

**False Positives.** Most groups are incorrectly classified due to the lack of capability to find functionally identical code clones in the clone detector we used.

We investigate the implementations of such clone groups and find that these groups can be reorganized by common functionality. For example, CI1, CI2, CI3, and CI4 are reported in the same group. A pair of CI1 and CI3 and a pair of CI2 and CI4 share the same features, respectively, but the former pair has the type variation and the latter one has the method call variation. Each reorganized group may be classified depending on the variations between counterparts. This limitation can be overcome by plugging in clone detectors that are more resilient to differences in syntax [38, 18].

**False Negatives.** We inspected the groups not detected by PRI. We found that some clone instances in a group can be reorganized as a sub-group, and other clone siblings can be moved to another sub-group, which causes false negatives. Figure 5.1 exemplifies the false negatives. As a clone detector reports six clone instances as a group, PRI tracks the change history and detects the group as unrefactored. However, partitioning based on program semantics can allow PRI to detect three unrefactored sub-groups: a set of AbstractAjpProtocol.pause and AbstractHttp11Protocol.pause, a set of AbstractAjpProtocol.resume and AbstractHttp11Protocol.resume, and a set of AbstractAjpProtocol.stop and AbstractHttp11Protocol.stop. These missing groups are hard to detect using PRI since our current templates are not capable of partitioning or merging clone groups to detect refactorable subgroups or most common groups. Heuristics to consider the scenario are planned as future work.
5.2 Case Studies for Fault Localization of Failure-Inducing Refactorings

5.2.1 Experimental Design for RQ3 and RQ4

We apply \textsc{Rit} to three subject applications: Log4J—a Java-based logging library, Apache Tomcat—a Java Web application server, and JRuby—an implementation of Ruby on the JVM.

We use a data set of over 100 refactoring transformations with seeded anomalies. Each transformation is a pair \((p_1, p_2)\) of Java programs, where both \(p_1\) and \(p_2\) produce no compilation error and \(p_2\) contains at least one seeded refactoring anomaly that breaks the behavior preservation of \(p_1\). We include four categories of seeded anomalies; (1) \(ME\) denotes missing edits that required edits are skipped during refactorings; (2) \(EE\) means extra edits that additional changes are added after refactorings; (3) \(LC_a\) denotes anomalies by introducing overriding methods during refactorings, and; (4) \(RC_a\) denotes anomalies by introducing overloading methods during refactorings. These anomalies are defined as variations from the guidelines of Fowler’s refactoring mechanics [17] by deliberately performing random changes between refactoring steps.

These faults are collected by the authors experienced when performing and inspecting refactorings, and they are examined based on previous studies that identified issues in automated refactorings [71, 11, 17]. The data set is assembled by the first author and later revised by the remaining authors.

Prior empirical studies on refactoring have used similar methodologies to inject anomalies [6, 12, 20, 21, 70]. For example, Gligoric et al. systematically apply refactorings at a large number of locations in open source projects and investigate failures caused by refactorings or refactoring-related compilation errors [21]. Table 5.4 shows specific change categories of seeded anomalies.

We apply five refactorings and three composite refactorings with the distribution of seeded anomalies per refactoring type such as 161 for \textit{Extract Method}, 10 for \textit{Pull Up Method}, 28 for \textit{Extract Super Class}, 4 for \textit{Rename Method}, 22 for \textit{Add Parameter}, 68 for \textit{Extract & Move Method}, 135 for \textit{Extract & Pull Up Method}, and 1 for \textit{Move & Rename Method}.\(^3\)

For the studies, we develop a mining strategy to obtain the ground truth data set

\(^3\)The summation of applied refactorings can be greater than the total number of refactoring transformations since one refactoring transformation can consist of multiple refactoring types such as Extract Method and Pullup Method.
Anomaly Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME</td>
<td>The return value is ignored at a call to the refactored method.</td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>A call to the refactored method is missed.</td>
<td></td>
</tr>
<tr>
<td>EE</td>
<td>The return value is changed at the refactored method.</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>The argument order of a call to the refactored method is changed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The argument value of a call to the refactored method is changed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The method name of a call to the refactored method is changed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The operator in the if conditional expression is changed in the refactored method.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A refactoring is applied to wrong code location(s).</td>
<td></td>
</tr>
<tr>
<td>LC_a</td>
<td>The program semantics are changed by introducing overriding method during a refactoring.</td>
<td>20%</td>
</tr>
<tr>
<td>RC_a</td>
<td>The program semantics are changed by introducing overloading method during a refactoring.</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 5.4: The categories of seeded refactoring anomalies, consisting of four major categories such as ME (20%), EE (40%), LC_a (20%), and RC_a (20%) from the three subject applications. We classify individual regression tests and check whether they address multiple development issues (i.e., composite tests). To mine composite tests, we write a program which takes as input call graphs of each test and the refactored locations, and then identify tests that includes a call to the refactored method and more than one of asserts. We manually partition these composite tests into test-slices. The manual inspection results are examined to filter out false positive composite tests that are mined. The data set was inspected by the first author and later reviewed by other authors.

In our data set, each refactoring transformation is a pair of \((p, p')\) of a program, where if a call graph of a test-slice is involved in refactorings in \(p'\), we add this test-slice as an affected test in our data set \(G_1\) to evaluate \(RQ1\). If any atomic change of a refactoring in \(p'\) is associated with affected test-slices, we add this refactoring as an affecting change in our data set \(G_2\) to evaluate \(RQ2\).

Based on the ground truth data set \(G_1\), precision \(P_1\) and recall \(R_1\) are measured as \(P_1 = \frac{|G_1 \cap S|}{|S|}\), \(R_1 = \frac{|G_1 \cap S|}{|G_1|}\), where \(P_1\) is the percentage of our identification result that are correct, \(R_1\) is the percentage of correct identification that RIT report, and \(S\) denotes the affected tests identified by RIT, all tests of which are automatically
partitioned into a set of test-slices. Based on the ground truth data set \( G_2 \), precision \( P_2 \) and recall \( R_2 \) are measured as \( P_2 = \frac{|G_2 \cap C|}{|C|} \), \( R_2 = \frac{|G_2 \cap C|}{|G_2|} \), where \( P_2 \) is the percentage of our detection result that are correct, \( R_2 \) is the percentage of correct detection that \( \text{RIT} \) report, and \( C \) denotes the affecting refactoring changes detected by \( \text{RIT} \), some atomic changes of which are automatically analyzed by change dependences.

This experiment was conducted on a machine with a quad-core 2.2GHz CPU and 16GB RAM.

5.2.2 Study Results and Discussion

Table 5.5 and Table 5.6 summarizes our evaluation result, to answer the questions raised above. We run \( \text{RIT} \) on each pair of the original and refactored program in our data set. We collect results of affected test-cases and affecting refactoring changes that \( \text{RIT} \) has identified, and manually validate their outcomes to collect precision, recall, and accuracy. During this validation process, the first author analyzed results, and then the results were validated in the meetings with the remaining authors. When there were disagreements, each case was discussed in a second analysis phase and a joint decision was made.

RQ3. Can \( \text{RIT} \) accurately determine test-slices affected by refactorings? We assess \( \text{RIT} \)'s precision by investigating how many of test-slices affected by atomic changes of refactorings are indeed true affected test-slices. \( \text{RIT} \) determines 394 affected test-slices, 343 of which are correct, resulting in 80.0% precision. Regarding recall, \( \text{RIT} \) identifies 82.1% of all ground truth data sets. It identifies 343 out of 388 test-slices. It determines affected test-slices, partitioning composite tests with 80.9% accuracy.

\( \text{RIT} \) determines test-slices affected by refactoring changes which are not easy to identify because these test-slices require regression fault analysis, investigating the impacts of individual atomic changes of refactorings, e.g., a refactoring application consisting of structural changes such as CM, DM, LC, RC, etc. Instead of decomposing composite tests, simply investigating independent development issues with multiple asserts often leads to difficulty validating refactoring anomalies by analyzing change dependencies [37].

False Positive. Test slices that are false positives are mostly due to API minimization issues during the composite test decomposition. For example, a composite test \( \text{JRubyEngineTest.testCall()} \) contains three assert statements in JRuby (r1.7.3). \( \text{RIT} \) correctly identifies the test \( \text{JRubyEngineTest.testCall()} \) affected by refactoring changes and decomposes this composite test into three test-slices based
Table 5.5: Accuracy of \textsc{RIt}'s capability to identify affected test-slices and detect failure-inducing refactorings. \textsc{PRJ} an evaluated project at a particular revision (\(L\) for Log4j, \(J\) for Jruby and \(T\) for Tomcat), \(A_{T}\) the number of affected failed tests in ground truth, \(A_{S_{g}}\): the number of affected failed test-slices in ground truth, \(A_{T}\): the number of affected failed tests identified by \textsc{RIt}, \(A_{S_{c}}\): the number of total test-slices identified by \textsc{RIt}, \(A_{S_{x}}\): the number of affected failed test-slices correctly identified by \textsc{RIt}, \(A_{t}\): the number of unrelated test-slices of failed test(s), \(P_{1}\): precision of affected test identification(\%), \(R_{1}\): recall of affected test identification(\%), \(A_{1}\): accuracy of affected test identification(\%), and each line represents the evaluation result for a project at a particular revision where manual refactoring changes have been conducted.

on the constructed program dependence graph. However, some domain-specific APIs (e.g., \texttt{BSFManager.registerScriptingEngine()}), which are required for JRuby’s resource management engine, are missed from each test-slice. The manual investigation for adding such API names in the \textsc{RIt}'s configuration can remove these issues but automated heuristics to consider a data reachability algorithm for third-party APIs are planned as future work [84].
### Table 5.6: Accuracy of RIT’s capability to identify affected test-slices and detect failure-inducing refactorings.

<table>
<thead>
<tr>
<th>ID</th>
<th>PRJ</th>
<th>AR</th>
<th>AR_u</th>
<th>AR_c</th>
<th>AR_x</th>
<th>P_2</th>
<th>R_2</th>
<th>A_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L-2.0</td>
<td>4/187</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L-2.1</td>
<td>8/10</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>L-2.2</td>
<td>4/48</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>L-2.3</td>
<td>38/342</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>L-2.4</td>
<td>1/4</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>L-2.5</td>
<td>9/90</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L-2.6</td>
<td>18/288</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>L-2.7</td>
<td>14/140</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>L-2.8</td>
<td>42/310</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>J-1.0</td>
<td>28/544</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>J-1.1</td>
<td>5/37</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>J-1.2</td>
<td>5/72</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>J-1.3</td>
<td>10/80</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>J-1.4</td>
<td>3/879</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>J-1.6</td>
<td>9/90</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>J-1.7</td>
<td>18/288</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>J-1.8</td>
<td>14/140</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>J-1.9</td>
<td>42/310</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>J-1.10</td>
<td>28/544</td>
<td>0/0</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**False Negative.** Our partial PDG construction prevents RIT from identifying some failed test slices. As RIT relies on the output of a static analysis tool Eclipse JDT (eclipse.org/jdt), it is hard to find the relevant data flow relationships beyond the method level. For example, RIT cannot find a failed test-slice of the test method T4 in Apache.
Log4j (r2.8) since it is not affected by the refactoring anomaly due to the incompletely constructed test-slice. Although **RIT** computes the complete PDG within the method $T$ to group relevant statements and predicates, it misses a method invocation statement, which affects the behavior of a preceding statement outside the method $T$. Although we design **RIT** to avoid unwieldy test-slices primarily caused from *overly broad definition of relevance* [72], checking the correctness of the test-slice construction to determine whether expanded dependences are uncovered for test-slices is our future work.

**RQ4.** Can **RIT** accurately detect failure-inducing refactoring edits? We estimate the precision of **RIT** by examining how many of the affecting refactoring edits are indeed a true refactoring anomaly. As **RIT** detects refactorings in which some atomic changes do not apply behavior-preserving code transformations, we consider any atomic change affecting the semantics of the program as a true refactoring anomaly. **RIT** detects 377 failure-inducing refactorings, 370 of which are correct, resulting in 97.0% precision. Regarding recall, **RIT** detects 98.1% of all ground truth data sets. It detects 370 out of 384 failure-inducing refactorings based on partitioned test-slices with 97.2% accuracy.

**RIT** helps developers inspect real regression faults by reducing the number of affecting changes related to each test failure that may still too large for manual inspection. Given failed test-slices separated from others in composite tests, it automatically identifies a subset of changes responsible for potential refactoring anomalies. Localizing failure-inducing refactoring edits is not easy since fault localization usually requires executing large regression test suites, which is time-consuming to run.

**False Positive.** Most refactorings are incorrectly detected due to the lack of capability to analyze the behavior impact of test-slices. For example, **RIT** correctly computes test-slices by collecting related statements and control predicates for each test-slices (e.g., *ConcurrentLocalContextProviderTest.testGetRuntime()* in JRuby (r1.7.0)). For the detection of failure-inducing refactorings, **RIT** runs a set of entire test-slices, which reveals all failures. However, running some test-slices individually did not report associated failures. We found that the a method call from preceding test-slice have an impact on the current test-slice behavior, which causes false positives on the detection of affecting refactorings, finding irrelevant changes. Heuristics to consider the scenario are planned as future work.

**False Negative.** We inspected refactoring changes not detected by **RIT**. We found that some static method calls defined in the library in tests cannot be analyzed by PDG due to constant values on the method parameters, which causes false negatives. Missing such method calls that do not contain any variables dependent on ```
prevents the detection of affecting refactorings. For example, RIT identifies four test-slices by correctly partitioning the failed test JRubyEngineTest.testGetBindings() in JRuby (r1.7.0). Before conducting the detection of refactorings responsible for these failures, it minimizes each test-slice based on the data and control dependent analysis, where a static method call System.setProperty("org.jruby.embed.local-context.scope", "singlethread") is excluded, leading to the incorrect dynamic call graph generation due to the modified behavior. Other refactorings not detected by RIT are caused from RQ1’s false negatives. These missing refactoring changes are hard to detect using RIT since our current implementation of PDG are not capable of identifying the library call edges accurately. Our future work will consider calculation of a library call abstraction that can be applied in program slicing and dataflow analysis.
CHAPTER 6

CONCLUSION

In this research, we propose two techniques for inspecting and testing for refactoring changes. First, PRI analyzes how clone instances are refactored consistently (or inconsistently) with other siblings in the same group. To summarize clone refactorings and detect incomplete refactoring anomalies, PRI employs refactoring pattern templates and traces cloned code fragments across revisions. It further analyzes refactoring anomalies to classify if developers are not easy to remove them using standard refactoring techniques. It also provides a novel visualization tool to highlight refactoring edit histories and anomalies. Second, we propose RIT to validate refactoring changes. Given refactorings, it applies change impact analysis to determine tests that are affected by such changes. If a test fails due to refactoring anomalies, it identifies a subset of changes responsible for these anomalies. RIT partitions a composite test and groups semantically related statements dependent on each assert. It applies data flow tracking to determine test-slices that are more cohesive and self-contained with respect to the issue being addressed.

As future work for improving our approach and tool, we intend to (i) create pattern templates for more refactoring types; (ii) provide tool support for fixing incomplete clone refactorings; and (iii) implement checking operations to determine the correctness of extra edits to pure refactoring versions. We also plan to conduct user studies with both student and professional developers to improve the usability.
REFERENCES


[80] Y. Yu, J. Jones, and M. Harrold. An empirical study of the effects of test-


