AUTOMATED IDENTIFICATION AND REFACTORING OF CODE CLONES IN TEST SCRIPTS FOR ELIMINATING DUPLICATION

ABDULMECİT ŞAHİN

ÖZYEĞİN UNIVERSITY

JUNE, 2025

AUTOMATED IDENTIFICATION AND REFACTORING OF CODE CLONES IN TEST SCRIPTS FOR ELIMINATING DUPLICATION

by
Abdulmecit Şahin

Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in
Computer Science

Advisor: Prof. Hasan Sözer

Graduate School of Science and Engineering
Özyeğin University
İstanbul

June, 2025

AUTOMATED IDENTIFICATION AND REFACTORING OF CODE CLONES IN TEST SCRIPTS FOR ELIMINATING DUPLICATION

Approved by:

Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Assoc. Prof. Reyhan Aydoğan
Department of Artificial Intelligence and Data Engineering
Özyeğin University

Assoc. Prof. Ayşe Tosun Kühn
Department of Computer Engineering
Istanbul Technical University

Approval Date: May 13, 2025

To my beloved wife

DECLARATION OF ORIGINALITY

I hereby declare that I am the sole author of this thesis and that this is the true copy of my thesis, including the final revisions, approved by my thesis committee. All data and information have been obtained, produced, and presented in accordance with the rules of research ethics and principles of academic honesty. As required by these rules, to the best of my knowledge I have acknowledged ideas, thoughts, and any copyrighted material in accordance with the standard referencing rules. I certify that any part of this thesis has not been submitted for a degree or diploma in another educational institution.

Abdulmecit Şahin

ABSTRACT

Code clones in test scripts can significantly reduce maintainability. They can lead to duplication especially in ecosystems and product families where applications have commonalities in their test scenarios. Our goal is to eliminate duplication in test scripts for reducing their size and improving their maintainability. In particular, we focus on Python test scripts and propose an automated approach to identify commonalities, detect variations, and apply refactoring. We compare the effectiveness of various clone detection tools in identifying code clones at the function level in Python test scripts. We apply static analysis to detect variations and code dependencies among similar functions to steer the automated refactoring process. The refactoring approach involves extracting new functions or generalizing existing functions through parameterization. We validate our approach through case studies on one industrial and three open-source projects. Our findings indicate that automated refactoring can lead to a reduction of up to 8% in the size of test scripts. Automated detection and refactoring of code clones in test scripts can eliminate code duplication and reduce the script size. The proposed approach, validated through case studies, shows promising results and can be applied in various software ecosystems to improve test script maintainability.

Keywords: Test script refactoring, code clone detection, duplicate test code elimination, industrial case study

ÖZET

Test betiklerindeki kod klonları, sürdürülebilirliği önemli ölçüde azaltabilmektedir. Özellikle test senaryolarında benzerliklerin bulunduğu ekosistemlerde ve ürün ailelerinde bu durum, kod tekrarına yol açabilmektedir. Bu çalışmanın amacı, test betiklerindeki tekrarları ortadan kaldırarak bu betiklerin boyutunu küçültmek ve bakımını kolaylaştırmaktır. Özellikle Python test betiklerine odaklanılarak, ortaklıkların otomatik olarak tespit edilmesi, varyasyonların belirlenmesi ve refaktörizasyonun uygulanmasına yönelik bir yöntem önerilmektedir. Çeşitli klon tespit araçlarının, Python test betiklerinde fonksiyonel düzeydeki kod klonlarını tespit etme etkinlikleri karşılaştırılmıştır. Benzer işlevler arasındaki farklılıkları ve kod bağımlılıklarını belirlemek amacıyla statik analiz uygulanmış, böylece otomatik refaktörizasyon süreci yönlendirilmiştir. Önerilen refaktörizasyon yaklaşımı, yeni işlevlerin çıkarılması veya mevcut işlevlerin parametreleştirme yoluyla genelleştirilmesini içermektedir. Yöntemimiz, bir endüstriyel ve üç açık kaynaklı proje üzerinde yapılan vaka çalışmalarıyla doğrulanmıştır. Elde edilen sonuçlar, otomatik refaktörizasyonun test betiklerinin boyutunda %8'e varan bir azalma sağlayabileceğini göstermektedir. Test betiklerindeki kod klonlarının otomatik olarak tespiti ve refaktörizasyonu, kod tekrarını ortadan kaldırarak betik boyutunu azaltabilmektedir. Vaka çalışmaları ile doğrulanan bu yaklaşım, farklı yazılım ekosistemlerinde test betiklerinin bakımını iyileştirmek amacıyla uygulanabilir niteliktedir.

Anahtar Kelimeler: Test betiği yeniden yapılandırma, kod klonu tespiti, yinelenen test kodunun ortadan kaldırılması, endüstriyel vaka çalışması

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

AST	Abstract syntax tree
CPD	Copy/Paste Detector
KLOC	Thousands of Line of Codes
LOC	Line of Code
NiCad	Automated Detection of Near-Miss Intentional Clones
Simian	Similarity Analyser
TRADE	Test Refactoring and Automated Duplicate Elimination
IoT	Internet of Things

1. INTRODUCTION

Testing plays a crucial role in ensuring the reliability of software systems. However, the associated costs can be substantial, particularly when high levels of reliability are required for large and complex systems under test. It is known that testing activities can constitute at least half of the overall development costs [2, 3]. A commonly employed strategy to reduce these costs is the adoption of test automation [4, 5]. The widespread use of test automation has increased especially with the adoption of continuous delivery and agile development practices.

Test automation can be applied at various levels, including unit, integration, and system testing. In this work, we study test scripts (test cases[1]) developed for automating system-level functional tests as well as those developed for unit testing [6]. These test scripts are subject to technical debt [7, 8, 9], which are sub-optimal solutions that negatively affect the comprehensibility, extensibility, and maintainability of test suites. Test smells [10] constitute a major indicator of this debt, including code duplication. We focus on the identification and elimination of duplicated code, which is caused by code reuse at different places of the script although the copied code snippets should be unified in a reused function [9].

Empirical studies show that the ratio of duplication in testing code can be up to 29% [1] and it can have strong negative impact on maintainability [10]. 41% of the participants in a recent survey consider removal of duplicated code as the main benefit of refactoring performed in testing code [11]. Moreover, the same survey identifies the high effort required and the lack of adequate tool support as the primary challenges hindering the adoption of this refactoring process. Our study aims to address these challenges.

In general, code duplication is considered harmful as it causes bug propagation

[1]Throughout this thesis, the terms *test case* and *test script* are used interchangeably, as it is assumed that each test case is implemented as a separate test script.

and increased maintenance efforts [12]. Therefore, software clone detection has been an active research area for decades. Several approaches have been introduced for the detection of code clones [12, 13], including text-based, token-based, tree-based, metric-based, semantic and hybrid approaches [14]. We aim at utilizing clone detection tools for the identification of common test scenarios and the associated duplicate testing code. In particular, we use tools that employ text-based, token-based and tree-based methods. Studies on code clones and clone detection tools are mostly focused on production code rather than testing code [1]. A recent study presents an empirical evaluation of clone detection in testing code [1] developed with Java. A year later, another study [15] revealed that test code contains twice as many clones as the production code. Their most recent study on the subject [16] emphasizes the need for specialized tools to detect, monitor and manage test clones.

In this study, we focus on testing code developed with Python, which is increasingly adopted for developing test scripts. We use clone detection to identify commonalities and code duplication in test scripts. Then, we automatically refactor the test code to eliminate duplication and reduce the test script size. In particular, we use a tool to find clones at the function level. Then, we employ static analysis to detect variations and data dependencies. Analysis results are used for either unifying a set of existing functions into parametrized and generalized versions or identifying parts of the functions that can be extracted as new functions. As a result, we introduce extracted functions reused by multiple other functions or generalized functions replaced with multiple similar ones.

We conducted case studies with test suites that include Python scripts developed for testing four different applications. One of these is developed for testing Smart TV systems currently being developed and maintained by a consumer electronics company. The other three are open source projects, one of which is a home automation system. The other two are cross-platform frameworks for developing graphical user interfaces. Our

results obtained with these four experimental objects show that up to 8% reduction in the size of test scripts is possible with automated refactoring.

Our contributions are threefold in this paper. Firstly, we present an empirical evaluation of clone detection tools regarding their effectiveness in detecting clones in Python test scripts. Secondly, we introduce an automated refactoring approach. Finally, we conduct case studies on testing code developed for three open-source applications and an industrial Smart TV system, offering an empirical analysis of the impact of code clones and potential improvements in their elimination and code size reduction.

The remainder of this paper is organized as follows. We summarize related studies in the following section.We present our overall approach in Chapter 3. In this section, we also introduce the set of experimental objects, which are used both for the comparison of a set of clone detection tools, and the evaluation of the effectiveness of our approach in eliminating duplication. In Chapter 4, we evaluate the effectiveness of clone detection tools applied on Python test scripts. In Chapter 5, we explain the refactoring process and present an evaluation of it. We discuss the results in Chapter 6. Finally, we conclude the paper in Chapter 7.

2. RELATED WORK

Numerous clone detection methods, techniques and tools have been proposed and evaluated in various contexts [14]. It was shown that there are differences between the sensitivity, recall rates and performance of different clone detection tools [17]. These tools have been applied both within individual codebases and across multiple projects [18], though they have primarily focused on production code rather than test code [19]. There are only a few studies that focus on testing code. An industrial case study [20] using the Automated Detection of Near-Miss Intentional Clones (NiCad) [21] tool confirmed that 49% of testing code consists of clones. More recently, a study [1] compared alternative clone detection tools, and reported that duplication in testing code ranges between 23% and 29%. That study was based on a dataset of open-source test suites developed in Java. In contrast, our work focuses on testing code developed in Python including both industrial and open-source projects. Moreover, we go beyond just the detection of clones by considering their potential for refactoring and proposing an automated approach for the refactoring process.

There are also studies that consider refactoring and removal of code clones rather than focusing on their detection only [22]. One study [23] analyzes the impact of clones on software quality metrics and shows that refactoring and elimination of clones can lead to improvements, especially in the cohesion, complexity and connectivity metrics. Another large-scale case study [24], involving 9 open-source projects and 4 clone detection tools, investigated the refactorability of code clones. While this study did not specifically target duplication in testing code, it found that testing code tends to contain more clones than production code. Another tool [25] aims at providing visualization support and indicates whether the detected clones can be safely refactored. Using this tool alongside CloneDR [26], a study [27] explored the detection and refactoring of function clones.

These studies mainly target clones in Java code with one exception [28], which investigates a geographical information system developed in C. All of them focus on clones in production code. In contrast, one of the studies [29] focuses on testing codes in Java-based open source projects and measures the effect of clone refactoring on the size of unit tests developed for object-oriented software. The study demonstrates how clone restructuring can reduce complexity by making unit tests smaller in size.

There also exist tools developed for automated refactoring of clones. JDeodorant [30] imports and analyzes data from tools that detect clones in Java Language. Then, it determines whether the clones are refactorable or not. Here, it consults the user and applies different levels of refactoring (Extract Method, Pull Up Method, Utility Method). Anticopypaster [31], on the other hand, does not need to manually run clone detection tools, but analyzes the code clones when the copy-paste operation occurs and offers Extract Method refactoring, if it is found to be appropriate. CRec [32] analyzes the history and current status of code clones to provide refactoring recommendations with machine learning. All of these tools are written for the analysis and refactoring Java code and they are all tested on production code.

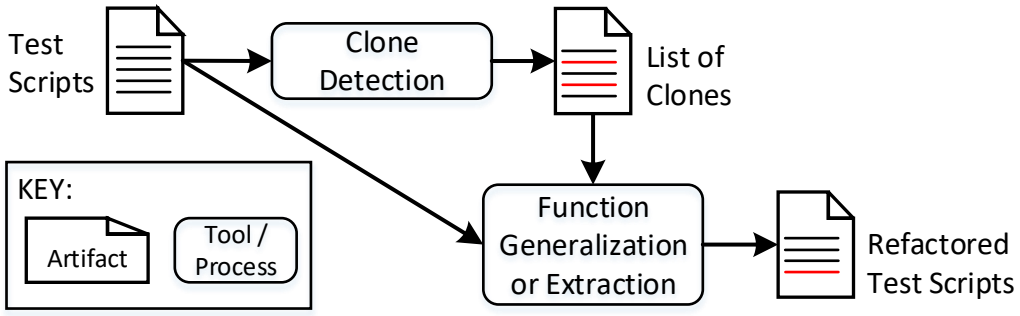
In this work, we evaluate alternative clone detection tools on Python test scripts using a dataset comprising both open-source and industrial test suites. Despite the widespread use of Python in test automation, no similar comparison has been conducted for either open-source or industrial projects. Some of the previous studies for Python examined code smells rather than code clones [33, 34]. Unlike previous studies, our research not only focuses on analysis but also extends to synthesis. We leverage clone detection tools to identify duplication in testing code and introduce automated refactoring techniques supported by static analysis to partially eliminate this duplication. RIdiom [35] focuses on refactoring of Python code, but instead of eliminating code clones, it aims at transforming non-idiomatic Python code into idiomatic Python code.

3. THE OVERALL APPROACH AND EXPERIMENTAL OBJECTS

The overall approach is presented in Figure 3.1. It involves two main steps. In the first step, the input test scripts are analyzed to identify the list of code clones. In the second step, a subset of the analyzed code snippets are redefined as generalized functions that are called from multiple locations. The candidate code snippets for refactoring, as well as the structure of the extracted functions are determined based on the levels of dependency and variability of the analyzed code snippets. The output of the approach is the refactored version of the input test scripts, where some of the duplicated code snippets are unified in a set of parametrized functions. We rerun the refactored test scripts on the same version of the source code to validate the accuracy of the refactoring process.

Both steps depicted in Figure 3.1 are automated for code clones at the function level. We utilize an existing tool to automate the first step. For this purpose, we evaluate and compare the effectiveness of alternative code clone detection tools in identifying code duplication that can be potentially eliminated by refactoring. This evaluation is presented in the following section (Section 4). This is followed by the explanation and evaluation of our refactoring approach, which includes variability analysis and function extraction (Section 5). Our approach is implemented in TRADE (Test Refactoring and Automated

Figure 3.1: *The overall approach.*



Duplicate Elimination), an open-source toolset available online in a public repository[1]. We use the same set of test suites for both the evaluation of the clone detection tools and the evaluation of TRADE as listed in Table 3.1.

Table 3.1: *Test suites that are used as experimental objects.*

Project	**Test Type**	**# of Files**	**# of Line of Code (LOC)**
Home Assistant	Unit	4,267	803,298
	Functional	551	57,463
Kivy	Unit	59	8,282
	Functional	28	2,622
Toga	Unit	70	15,271
	Functional	75	9,256
Smart TV	Functional	26	8,224

The last project listed in Table 3.1 is an industrial project on *Smart TV* systems. We used system-level functional test scripts developed for this project, which is developed and maintained at one of the largest consumer electronics companies in Europe, Vestel[2]. In addition to the products of its own, Vestel is producing Smart TV systems for 157 different brands from 145 different countries [36, 37]. The first three projects listed in Table 3.1 are open-source projects. The first one is *Home Assistant*[3], which allows the use of multiple, communicating Internet of Things (IoT) devices. *Kivy*[4] and *Toga*[5] are cross-platform frameworks implemented with Python for developing graphical user interfaces. We used test suites from all three projects, each of which includes both unit and system-level functional tests. Since open source projects are constantly changing and updated, we captured a snapshot of their test suites for the repeatability of our experimental evaluation.

[1] `https://github.com/iOTMecit/TRADE`
[2] `http://www.vestel.com.tr`
[3] `https://github.com/home-assistant`
[4] `https://github.com/kivy`
[5] `https://github.com/beeware/toga`

In particular, we used version 2024.4.0 of the *Home Assistant* project, version 2.3.0 of the *Kivy* project and version 0.4.7 of the *Toga* project. The corresponding test suites as well as their refactored versions are all available at the TRADE repository. We cannot share the dataset regarding the *Smart TV* project due to confidentiality.

All test scripts collected from the four projects were developed in Python. In the following section, we present an empirical evaluation and comparison of code clone detection tools applied to these Python test scripts.

4. CLONE DETECTION IN PYTHON TEST SCRIPTS

Code clones refer to segments of code that exhibit similarity based on a defined criterion [26]. These clones typically arise from reusing code fragments, with or without slight modifications. They can decrease maintainability and cause bug propagation. It was shown that testing code can contain more than twice as much duplication as production code [38], which increases the impact of cloning on the maintenance of test suites.

As the size and complexity of software projects increase, manually analyzing the similarities and differences between code segments becomes impractical and expensive. This makes automated clone detection crucial, and it has been an active area of research for decades [12]. Numerous techniques and tools have been developed and evaluated for the detection of various types of code clones [12, 13]. In general, test clones are categorized into 4 types. Type 1 clones are exact copies with only whitespace or comments altered. Type 2 clones are syntactically identical, with changes limited to identifiers like variables or function names. Type 3 clones involve modifications, additions, or deletions of statements. Finally, Type 4 clones have similar semantics implemented differently. The type as well as the granularity level (e.g., statement, code block, function) of the detected clones can vary. Yet another variation point among the proposed tools is the employed approach for clone detection, which include text-based, token-based, Abstract syntax tree (AST)-based, metric-based, semantic and hybrid approaches [14]. Therefore, we would like to address the following two questions before evaluating TRADE itself:

- How effective are alternative tools in detecting code clones, specifically within Python test scripts?
- To what extent do alternative tools support the modification of test scripts by providing output on code clones in terms of similarity levels, clone types, and granularity for identifying refactoring opportunities?

In the following, we first introduce the set of clone detection tools we evaluated in this study and their properties. Then, we present an evaluation based on our experimental objects (See Table 3.1).

4.1 Evaluated Clone Detection Tools and Settings

We evaluated three clone detection tools listed in Table 4.1: NiCad [21], Copy/-Paste Detector (CPD) [39], and Similarity Analyser (Simian) [40], all capable of detecting clones in Python code. All of these tools are freely available and they support multiple platforms. They also support multiple programming languages but we are interested in their ability to detect clones in Python code in this study. Table 4.1 lists the clone types, adopted clone detection approach and the granularity of detected clones for the evaluated tools.

Table 4.1: *Evaluated clone detection tools.*

Tool	Clone Types	Approach	Granularity
NiCad	1, 2, 3	AST-based	block, function, file
CPD	1, 2	token-based	block
Simian	1	text-based	block

We can see in Table 4.1 that NiCad supports the detection of a wider range of clone types and granularity levels. The capabilities of these tools differ for other programming languages. For instance, Simian can detect Type 2 clones in Java and C code. CPD can detect Type 2 clones although its support for this clone type is explicitly documented only for Java and C++ code.

As shown in Appendix A, the tools produce various outputs. In all three tools, clones are reported based on the minimum number of common lines considered as a threshold. This threshold parameter was set to 5 for all three tools, meaning code fragments are identified as clones only if they share at least 5 lines. CPD performs a token-based analysis for clone detection and the corresponding threshold parameter is set for the

number of tokens instead of number of lines. We set this parameter as 35, approximately corresponding to 5 lines. The reasoning behind this threshold is that refactoring a clone into an external, reusable function typically requires at least 4 additional lines of code to handle variations, import the newly generated function, and call it. As a result, refactoring code snippets shorter than 5 lines is not worthwhile. NiCad uses an additional threshold parameter called the similarity rate. We set this parameter as 0.3, meaning that detected clones should have minimum 70% similarity. We calculated the number of clone pairs detected by each tool. This number is already provided by NiCad. On the other hand, Simian and CPD provides clone groups. We calculated the number of clone pairs for each such group as $n(n-1)/2$, where n is the number of clones within the group.

Table 4.2: *Clone detection results for the test suites of each project.*

Project Name	**Tool Name**	**Clone Granularity**	**Clone Classes**	**Clone Pairs**
Home Assistant	**Simian**	Block	389	2,732
	CPD	Block	184	1,316
	NiCad	Block	111	1,028
		File	30	683
		Function	107	982
Kivy	**Simian**	Block	112	292
	CPD	Block	116	363
	NiCad	Block	57	258
		File	0	0
		Function	55	305
Toga	**Simian**	Block	195	1,643
	CPD	Block	154	523
	NiCad	Block	85	589
		File	4	5
		Function	100	568
Smart TV	**Simian**	Block	74	5,115
	CPD	Block	89	5,739
	NiCad	Block	34	3,018
		File	1	10
		Function	26	867

4.2 Clone Detection Results

We present the results in the Table 4.2. These results were obtained after a pre-processing applied on test scripts and post-processing applied on clone detection reports. We observed that NiCad could not parse some files due to syntax errors (TXL0192E, TXL0127E). We also observed that Simian and CPD tend to report the same clone multiple times. We developed a script that removes the files that are subject to parsing errors and another script that removes duplicate clone reports to obtain unique clone groups. When analyzing the results without excluding any files or clone reports, Simian detected 29,144 clone classes and 697,438 clone pairs in the *Home Assistant* project, whereas CPD identified 28,310 clone classes and 670,652 clone pairs. On the other hand, NiCad identified 111 clone classes and 1,028 clone pairs at the block level, 30 clone classes and 683 clone pairs at the file level, 107 clone classes and 982 clone pairs at the function level.

We manually examined the clone detection reports of each tool to understand the reasons behind duplicate reports and high variance in the number of clones reported by the tested tools. Since the CPD tool is token-based, it performs clone detection by reducing tokens. For example, after finding a clone between Python files as **`a.py`** and **`b.py`** with ***n*** tokens, it performs a new check by reducing the number of tokens to ***n-1***. If at this stage it finds a match with another file (e.g. **`c.py`**), it marks **`a.py`**, **`b.py`** and **`c.py`** as clones of each other. This approach results in multiple clone classes on the same lines, inflating the number of reported clones. We observed that the clustering capability of NiCad tool is very effective for avoiding the repetition of clones.

Since Simian and CPD tools can detect clones at the block level only, we compared the three tools at this granularity level. Table 4.3 lists results for different clone types detected at the block level. When we examined the results of Simian and CPD, we saw that Simian gives only Type 1 clones, while CPD gives a mix of Type 1 and Type 2 clones. Since we can get Type 1 and Type 2 clone outputs separately in NiCad, we took

these outputs and compared the Type 1 results with Simian and the blended Type 1 + Type 2 results with CPD.

Table 4.3: *Number of clones of different types at the block level.*

Project Name	Tool Name	Clone Type			
		Type 1		Type 1 + Type 2	
		Clone Classes	Clone Pairs	Clone Classes	Clone Pairs
Home Assistant	Simian	389	2,732	NA	NA
	CPD	NA	NA	184	1,316
	NiCad	22	33	91	381
Kivy	Simian	112	292	NA	NA
	CPD	NA	NA	116	363
	NiCad	1	6	24	55
Toga	Simian	195	1,643	NA	NA
	CPD	NA	NA	154	523
	NiCad	10	14	64	126
Smart TV	Simian	74	5,115	NA	NA
	CPD	NA	NA	89	5,739
	NiCad	26	455	79	1,445

4.3 Evaluation and Comparison

4.3.1 Effectiveness

Our findings align with previously reported literature, which emphasizes the significant quantity and density of code clones present in test scripts. Existing tools are also highly effective in detecting these clones in Python test scripts. We used the *precision* and *relative recall* metrics that are defined by Bladel and Demeyer [1] for a quantitative comparison among the tools. Their usage is as follows:

$$precision = \frac{c_{TP} * 100}{c_{all}}$$

, where c_{TP} denotes the number of true positive clones found by the tool and c_{all} the total number of clones found by the tool.

$$Recall = \frac{c_{all} * 100}{c_{tot}}$$

, where c_{all} denotes the total number of clones found by the tool and c_{tot} the total number of clones in the dataset. Since the exact number of clones in the dataset is not known,

Bladel and Demeyer estimate c_{tot} by aggregating all clones identified by the four tools. This estimation is commonly referred to as *relative recall*.

Table 4.4 presents the number of code clones detected exclusively by each tool and those detected commonly by both tools, for Type-1 and combined Type-1+Type-2 clones across four different projects. For example, in the Type-1 section, *Simian-Only* refers to the number of clones detected only by Simian, whereas *Common* is the number of clones reported by both Simian and NiCad. Since Simian also includes comment lines in the start and end line numbers of clone classes, a tolerance of 5 lines was applied when comparing clone class positions reported by both tools. In other words, two clone classes are considered to match if they belong to the same file and their start and end lines differ by no more than 5 lines.

This data can also be used to compute the *relative recall* of each tool. For instance, the relative recall of Simian for Type-1 clones can be calculated as:

$$\text{Relative Recall}_{\text{Simian}} = \frac{\text{Simian-Only} + \text{Common}}{\text{Simian-Only} + \text{NiCad-Only} + \text{Common}}$$

A script was written to calculate the number of common clones between two tools. You can refer to Appendix C for examples of clones that are detected, not detected and commonly by different clone detection tools.

Table 4.4: *Number of clones detected exclusively by each tool and commonly by both tools for Type-1 and Type-1+Type-2 clones across projects*

Project Name	Type 1			Type 1 + Type 2		
	Simian-Only	Nicad-Only	Common	CPD-Only	Nicad-Only	Common
Home Assistant	378	11	11	169	76	15
Kivy	111	0	1	115	23	1
Toga	191	5	5	148	58	6
Smart TV	56	8	18	65	55	24

Results are listed in Table 4.5. We see that the precision is perfect for all the tools and projects. There are no false positive reports. We see that relative recall for Simian and CPD are better than those achieved by NiCad. However, note that this measure does not reflect the actual recall for which we do not have a ground-truth.

Table 4.5: *Clone detection performance according to the precision and relative recall metrics as defined by Bladel and Demeyer [1].*

Project Name	Precision				Relative Recall			
	Type 1		Type 1 + Type 2		Type 1		Type 1 + Type 2	
	Simian	NiCad	CPD	NiCad	Simian	NiCad	CPD	NiCad
Home Assistant	100%	100%	100%	100%	97.0%	5.5%	92.5%	45.7%
Kivy	100%	100%	100%	100%	100%	0.9%	99.1%	20.5%
Toga	100%	100%	100%	100%	98.0%	5.0%	96.3%	40.0%
Smart TV	100%	100%	100%	100%	80.4%	28.3%	78.8%	70.0%

Although NiCad's relative recall performance is lower compared to the other two tools, our observations show that it detects clones at a higher level of granularity. This results in fewer detected clones but offers more meaningful insights for refactoring opportunities. Clones identified by NiCad can be split into smaller, less coherent segments, each of which can be considered as different clones by CPD and Simian. This, in turn, increases the number of clones detected although the same clone might be listed in several different combinations and locations. This fragmentation can obscure the true extent of code duplication and make it more challenging to apply refactoring to eliminate duplication.

4.3.2 Modifiability

We can see an example pair of code snippets in Figure 4.1. Hereby, the whole code snippets depicted on the left and right sides of the figure are reported as clones by NiCad. The NiCad tool reported the entire code snippet as a clone, while Simian reported the code lines in red color (Lines 6-12) as a clone, and CPD reported the code lines in red and blue colors (Lines 5-12) as a clone. Other appearances of these sections in other locations are reported as clones as well.

Figure 4.1: *Comparison of code clones detected by various tools (NiCad: Lines 1-21, Simian: Lines 6-12, CPD: Lines 5-12).*

```
"""Kivy/tests/test_properties.py:
    345-370"""

from kivy.properties import
    BoundedNumericProperty

bnp = BoundedNumericProperty(0, min=-5,
    max=5, errorvalue=1)
if set_name:
    bnp.set_name(wid, 'bnp')
    bnp.link_eagerly(wid)
else:
    bnp.link(wid, 'bnp')
    bnp.link_deps(wid, 'bnp')
bnp.set(wid, 1)
self.assertEqual(bnp.get(wid), 1)
bnp.set(wid, 5)
self.assertEqual(bnp.get(wid), 5)
bnp.set(wid, 6)
self.assertEqual(bnp.get(wid), 1)
bnp.set(wid, -5)
self.assertEqual(bnp.get(wid), -5)
bnp.set(wid, -6)
self.assertEqual(bnp.get(wid), 1)

```

```
"""Kivy/tests/test_properties.py:
    373-401"""

from kivy.properties import
    BoundedNumericProperty

bnp = BoundedNumericProperty(0, min=-5,
    max=5, errorhandler=lambda x: 5 if x
     > 5 else -5)
if set_name:
    bnp.set_name(wid, 'bnp')
    bnp.link_eagerly(wid)
else:
    bnp.link(wid, 'bnp')
    bnp.link_deps(wid, 'bnp')
bnp.set(wid, 1)
self.assertEqual(bnp.get(wid), 1)
bnp.set(wid, 5)
self.assertEqual(bnp.get(wid), 5)
bnp.set(wid, 10)
self.assertEqual(bnp.get(wid), 5)
bnp.set(wid, -5)
self.assertEqual(bnp.get(wid), -5)
bnp.set(wid, -10)
self.assertEqual(bnp.get(wid), -5)

```

In general, NiCad tends to group larger chunks of code into one clone and present it as a single block, simplifying the representation of cloned segments. This method is particularly beneficial in cases where small variations or slight differences exist within the code, as NiCad filters these out using its threshold settings. Rather than aggregating large chunks of code into one clone, Simian and CPD often break cloned sections into smaller parts, especially if these sections contain minor differences. Consequently, the

same code segment might appear as multiple smaller clones in Simian and CPD, whereas NiCad would group these segments into one larger clone. This can lead to an increased number of smaller clones in Simian and CPD, potentially making the clone detection output more granular but also harder to interpret at a high level.

The output of Simian and CPD can be more useful for manual refactoring of test scripts as discussed in Section 6.2. However, the extraction of functions at a higher granularity and parameterizing them for handling minor variations are more applicable for an automated approach. Therefore, in terms of their usability for TRADE, we find the output of NiCad more useful for identifying refactoring opportunities in test scripts. In the following, we explain the refactoring process and present an evaluation of the effectiveness of TRADE in refactoring Python test scripts.

5. AUTOMATED REFACTORING OF TEST SCRIPTS

We employed the output of NiCad at the function level to support function extraction and generalization. The threshold was set as 0.2 in order to increase the similarity of the reported clones that are suitable for refactoring operations. The number of detected clones are listed in Table 5.2 under the column titled *Before*, showing the results before the refactoring process. The refactoring process implemented in Test Refactoring and Automated Duplicate Elimination (TRADE) is outlined in Algorithm 1.

The refactoring process takes clone classes reported by NiCad as input. Each clone class, C represents a cluster of functions that are similar to each other and the level of similarity is quantified (i.e., $C.similarity$). There are two threshold values, *threshold1* and *threshold2* for the expected similarity levels provided as input. In our experimental setup, these values are set as 90 and 80 out of 100, respectively. The similarity level for each clone class, C is compared with respect to these thresholds. If the similarity level is above 90, functions in C are checked for class dependency (Line 5). If no dependency is found, one of the functions in C is taken as a reference function, *rf* (Line 6). Variations of *rf* are analyzed with respect to every other function in C (Lines 7-13). The function signature and the body is modified for every variation, s such that a new argument is added to be used within the function body (Lines 9-12). A scalar or variable can be replaced with an argument in cases of data dependencies. However, the statement itself may also vary, involving different computations or calls to different functions. These variations are managed by introducing an argument that encapsulates the entire corresponding statement and executing it within the function body using the *exec* function. The *exec* function in Python takes an expression as a string and dynamically executes it as Python code. Once all the variations are handled, the other functions in C other than *rf* are removed and all the calls to these functions are replaced with a call to *rf* (Lines 14-16).

If the similarity level for C is not above 90, but if it is above 80, then the code

Algorithm 1 The Automated Refactoring Process

```
1: Input: clone classes, threshold1, threshold2
2: procedure TRADE
3:     for each C ∈ clone classes do
4:         if C.similarity > threshold1 then
5:             if C has no class dependency then
6:                 rf ← a function ∈ C
7:                 for each function f ∈ C such that f ≠ rf do
8:                     vars ← statement variations (f, rf)
9:                     for each s ∈ vars do
10:                        Parameterize s as function argument arg_s in rf
11:                        Replace s in rf with exec(arg_s)
12:                    end for
13:                end for
14:                for each call c_f to function f ∈ C do
15:                    Import rf and replace c_f with a call to rf
16:                end for
17:            end if
18:        else if C.similarity > threshold2 then
19:            lines ← matching lines in C without class dependency
20:            if length(lines) > 5 then
21:                defvars ← variables defined in lines
22:                usevars ← variables used in lines
23:                Create a new function sf(usevars) with lines as the body
24:                Define variables in defvars as global variables
25:                for each f ∈ C do
26:                    Import sf and replace lines in f with a call to sf(usevars)
27:                end for
28:            end if
29:        end if
30:    end for
31: end procedure
```

lines that match among the functions are extracted (Line 18). Lines are selected as the matching lines in C that do not have class dependency (Line 19). It is checked that the number of lines remaining after separating the lines with class dependency is greater than 5 (Line 20). AST is analyzed to resolve data dependencies and determine the list of global variables and arguments for the function to be created. Hereby, two sets of variables are identified: *defvars*: variables that are defined within the matching lines and used later on (Line 21), *usevars*: variables that are defined before, and used within the matching lines (Line 22). A new function *sf* is extracted that includes the matching lines as the body (Lines 23). Variables in *usevars* are passed to the extracted function as arguments. Variables in *defvars* are defined to be global within the function (Line 24). Finally, all functions in C are modified such that the matching lines are replaced with a call to *sf* (Lines 25-27).

As the preliminary step in the refactoring process, function clones with 100% similarity are handled, if any. Since these clones represent functions that are identical, duplicate functions are removed and calls to the removed functions are replaced with a call to its clone. Then, clone classes with similarity 80% and above are processed as outlined in Algorithm 1. We applied this process to all the subject systems to answer the following questions:

- **RQ1**: To what extent can the reported code clones be refactored automatically?
- **RQ2**: What is the impact of automated refactoring on the reduction of code duplication and the size of testing code?
- **RQ3**: Is there a difference between functional tests and unit tests in terms of the amount of code duplication removed with automated refactoring?

The number of clones for which the refactoring was applied is listed in Table 5.1.

Table 5.1: *The number of refactored function clones by TRADE based on the similarity levels, and the reduction achieved in code size.*

Project / # of Clones	**100% Similarity**	**[90-100)% Similarity**	**[80-90)% Similarity**	**Reduction in the # of LOC**
Home Assistant	47	19	10	699
Kivy	0	1	9	24
Toga	31	19	18	640
Smart TV	8	5	7	670

Figure 5.1 shows a clone pair reported by NiCad with 93% similarity for the *Home Assistant* project, for instance. We can see that the two functions are identical except for the line 15.

Figure 5.1: *A clone detected by NiCad with 93% similarity in the* Home Assistant *project.*

```
1  """HomeAssistant/tests/components/sunweg
       /conftest.py: 54-72"""
2
3  def plant_fixture(inverter_fixture) ->
       Plant:
4   plant = Plant(
5    123456,
6    "Plant #123",
7    29.5,
8    0.5,
9    0,
10   12.786912,
11   24.0,
12   "kWh",
13   332.2,
14   0.012296,
15   datetime(2023, 2, 16, 14, 22, 37),
16  )
17  plant.inverters.append(inverter_fixture
       )
18  return plant
19
```

```
1  """HomeAssistant/tests/components/sunweg
       /conftest.py: 74-90"""
2
3  def plant_fixture_alternative(
       inverter_fixture) -> Plant:
4   plant = Plant(
5    123456,
6    "Plant #123",
7    29.5,
8    0.5,
9    0,
10   12.786912,
11   24.0,
12   "kWh",
13   332.2,
14   0.012296,
15   None,
16  )
17  plant.inverters.append(inverter_fixture
       )
18  return plant
19
```

Figure 5.2 shows the reference function and a refactored function that calls the reference function corresponding to the clone class with clone pairs shown in Figure 5.1. The only difference of the reference function with respect to the functions in Figure 5.1 is that the line 15 is parameterized. If there are more than two clone pairs in a clone class, the reference function is compared with all the other clone functions one by one. Each variance at a different line leads to a new parameter.

Figure 5.2: *The refactored versions of the functions shown in Figure 5.1.*

```
1  """HomeAssistant/tests/components/sunweg
       /conftest.py: 54-72"""
2
3  def plant_fixture(inverter_fixture,
       param1='datetime(2023, 2, 16, 14,
       22, 37),'):
4      plant = Plant(
5          123456,
6          "Plant #123",
7          29.5,
8          0.5,
9          0,
10         12.786912,
11         24.0,
12         "kWh",
13         332.2,
14         0.012296,
15         exec(param1)
16     )
17     plant.inverters.append(
       inverter_fixture)
18     return plant
19
```

```
1  """HomeAssistant/tests/components/sunweg
       /conftest.py: 74-90"""
2
3  def plant_fixture_alternative(
       inverter_fixture) -> Plant:
4      plant_fixture(inverter_fixture,
       param1='None,')
5
```

Figure 5.3 shows an example clone pair reported by NiCad with 80% similarity for the *Kivy* project. Recall from Algorithm 1 that TRADE looks for matching lines for such functions, and lines 6 to 15 in both functions are identical in this example pair. However,

statements listed in line 13 and thereafter have dependencies on the class (i.e., call to *self*) and as such cannot be extracted. As a result, lines 5-12 are extracted as Figure 5.4.

Figure 5.3: *A clone detected by NiCad with 80% similarity in the* Kivy *project, and the part that is extracted as a function (Lines 5-11), highlighted with red color.*

```
1  """Kivy/tests/test_properties.py:
       344-370"""
2  def test_bounded_numeric_
       property_error_value(self, set_name)
       :
3   from kivy.properties import
       BoundedNumericProperty
4   bnp = BoundedNumericProperty(0, min=-5,
        max=5, errorvalue=1)
5   if set_name:
6     bnp.set_name(wid, 'bnp')
7     bnp.link_eagerly(wid)
8   else:
9     bnp.link(wid, 'bnp')
10    bnp.link_deps(wid, 'bnp')
11  bnp.set(wid,1)
12  self.assertEqual(bnp.get(wid),1)
13  bnp.set(wid, 5)
14  self.assertEqual(bnp.get(wid),5)
15  bnp.set(wid, 6)
16  self.assertEqual(bnp.get(wid),1)
17  bnp.set(wid,-5)
18  self.assertEqual(bnp.get(wid),-5)
19  bnp.set(wid,-6)
20  self.assertEqual(bnp.get(wid), 1)
21
```

```
1  """Kivy/tests/test_properties.py:
       372-401"""
2  def test_bounded_numeric_
       property_error_handler(self,
       set_name):
3   from kivy.properties import
       BoundedNumericProperty
4   bnp = BoundedNumericProperty(0, min=-5,
        max=5, errorhandler=lambda x: 5 if
       x > 5 else -5)
5   if set_name:
6     bnp.set_name(wid, 'bnp')
7     bnp.link_eagerly(wid)
8   else:
9     bnp.link(wid, 'bnp')
10    bnp.link_deps(wid, 'bnp')
11  bnp.set(wid,1)
12  self.assertEqual(bnp.get(wid),1)
13  bnp.set(wid,5)
14  self.assertEqual(bnp.get(wid),5)
15  bnp.set(wid,10)
16  self.assertEqual(bnp.get(wid),5)
17  bnp.set(wid,-5)
18  self.assertEqual(bnp.get(wid),-5)
19  bnp.set(wid,-10)
20  self.assertEqual(bnp.get(wid),-5)
21
```

After the lines to be extracted are determined, the corresponding AST is parsed to identify dependencies on the context and data values. External values with data dependencies are added as arguments to the extracted function. Lines that have class dependencies

are not included in the extracted function to avoid contextual dependencies. The extracted function is copied at the end of the py file that includes the first function processed in the clone class. Hence, all the libraries and modules needed by the extracted function are already imported on top of the corresponding file.

Figure 5.4: *The function extracted after processing the clone pair shown in Figure 5.3.*

```
"""Kivy/tests/test_properties.py: 1333-1340"""


def extracted_function_1(set_name, wid, bnp):
    if set_name:
        bnp.set_name(wid, 'bnp')
        bnp.link_eagerly(wid)
    else:
        bnp.link(wid, 'bnp')
        bnp.link_deps(wid, 'bnp')
    bnp.set(wid, 1)

```

Figure 5.5 shows the pair of function clones listed in Figure 5.3, after they are refactored to invoke the extracted function shown in Figure 5.4 in place of the duplicated statements. We see that the variables used within the function are sent as an argument. If there are variables defined in the extracted function, these variables are automatically detected before the extraction and they are defined as non-local to be used outside the function. If a clone class contains more than a pair of functions, the extracted function is evaluated with respect to the other functions in the clone class as well. The duplicated code segments are replaced with a call to the extracted function. If the replacement is made in different py files, the py file containing the extracted function is imported into the corresponding py files.

Figure 5.5: *New versions of the original functions shown in Figure 5.3, which are refactored to invoke the extracted function shown in Figure 5.4 in place of the duplicated statements.*

```
"""Kivy/tests/test_properties.py:
    344-370"""

def test_bounded_numeric_property_
    error_value(self, set_name):
  from kivy.properties import
    BoundedNumericProperty
  bnp = BoundedNumericProperty(0, min
    =-5, max=5, errorvalue=1)

  extracted_function_1(set_name, wid,
    bnp)

  self.assertEqual(bnp.get(wid), 1)
  bnp.set(wid, 5)
  self.assertEqual(bnp.get(wid), 5)
  bnp.set(wid, 6)
  self.assertEqual(bnp.get(wid), 1)
  bnp.set(wid, -5)
  self.assertEqual(bnp.get(wid), -5)
  bnp.set(wid, -6)
  self.assertEqual(bnp.get(wid), 1)
```

```
"""Kivy/tests/test_properties.py:
    372-401"""

def test_bounded_numeric_property_
    error_handler(self, set_name):
  from kivy.properties import
    BoundedNumericProperty
  bnp = BoundedNumericProperty(
      0, min=-5, max=5,
      errorhandler=lambda x: 5 if x > 5
     else -5)

  extracted_function_1(set_name, wid,
    bnp)

  self.assertEqual(bnp.get(wid), 1)
  bnp.set(wid, 5)
  self.assertEqual(bnp.get(wid), 5)
  bnp.set(wid, 10)
  self.assertEqual(bnp.get(wid), 5)
  bnp.set(wid, -5)
  self.assertEqual(bnp.get(wid), -5)
  bnp.set(wid, -10)
  self.assertEqual(bnp.get(wid), -5)
```

We used the clone detection tools introduced in the previous section (i.e., Simian, CPD, NiCad) on the testing code of all the projects before and after the automated refactoring. Table 5.2 lists the results. The first column lists the projects. The second column lists the clone detection tools used. The third column list the granularity of clone detection. The last four columns lists the number of clone classes and the number of clone pairs

Table 5.2: *Comparison of clone detection results before and after automated refactoring.*

Project	Tool	Granularity	Clone Classes		Clone Pairs	
			Before	After	Before	After
Home Assistant	Simian	Block	29,144	29107	697,438	697,298
	CPD	Block	28,310	28,278	670,652	670,507
	NiCad[a]	Block	107	34	895	537
		File	23	12	490	475
		Function	112	27	683	400
Kivy	Simian	Block	182	176	412	400
	CPD	Block	241	232	683	633
	NiCad[a]	Block	51	34	156	101
		File	0	0	0	0
		Function	50	34	177	123
Toga	Simian	Block	558	513	3,119	2,667
	CPD	Block	458	388	1,497	1,549
	NiCad[a]	Block	84	16	407	69
		File	4	0	4	0
		Function	96	20	356	59
Smart TV	Simian	Block	129	130	6,178	6,222
	CPD	Block	285	281	11,832	10,724
	NiCad[a]	Block	49	23	2,437	1,346
		File	1	1	10	1
		Function	30	13	560	88

[a]Threshold defined as 0.2.

detected by these tool before the refactoring and after the refactoring. We can see that the both numbers are reduced after the refactoring, although the amount of reduction highly varies across projects. The highest improvement is achieved with *Home Assistant* project, partly due to the fact that its testing code was subject to the highest number of clones. We can also see some exceptional cases for the *Smart TV* project, where the number of clone classes (129) and the number of clone pairs (6,178) detected by the Simian tool are even increased (130 and 6,222, respectively) after refactoring. The reason for this result is the fine-grained clone detection of the tool, where very short snippets of code like function calls are also reported as clones.

In the following section, we further discuss the results, considering the reasons for testing code clones that cannot be refactored automatically, differences with respect to the type of testing performed, and the impact of refactoring on the testing code.

6. DISCUSSION

Automated refactoring could not be applied for all detected clones even if the reported similarity among them is high. In the following, we analyze these cases and the reasons for the limitations. Hereby, we also explore the potential improvements in testing code that could be achieved through manual refactoring based on the output of clone detection tools. This evaluation is conducted for all three clone detection tools, Simian, CPD, and NiCad. Then, in Section 6.3, we examine the impact of refactoring on testing code for different test types, namely unit tests and functional tests. Finally, we conclude the section in Section 6.4 with a discussion of the validity threats to our evaluation.

6.1 Automated Refactorability of Reported Code Clones and Challenges (RQ1)

After observing the effectiveness of automated refactoring in removing duplications and reducing the testing code size, we also investigated the full potential of refactoring when applied manually for detected clones in testing code. To assess the extent of improvements possible through manual refactoring, the first author of this paper tried to manually refactor the source code based on the reported clones. Due to the large number of clones reported by Simian and CPD in some projects, a random sample of 100 clones from the reports was selected for manual review. We observed that certain clone segments in the testing code could not be refactored, even manually, or that refactoring them offered no significant benefits in terms of code simplification or size reduction.

Table 6.1 summarizes the overall results. The reasons for not being able to refactor clone classes are categorized into four types as listed below. Illustrative examples for each category are provided in Appendix B.

- **Code Dependencies:** In many instances, clones were tightly coupled with other parts of the code, either through variable use, function calls, or interactions with

external modules. This makes it difficult to extract the cloned segments into a reusable function without breaking the dependencies. For example, in the *Home Assistant* project, 8 instances from the Simian reports could not be addressed due to such dependencies.

- **Lack of Similarity:** In some cases, similarities between the code segments were too superficial. While the clone detection tools identified the segments as clones, the actual logic or purpose of the code was different, making it impossible to refactor them into a single reusable function. This was observed primarily in the NiCad reports for *Home Assistant*, where 4 cases were marked not refactorable.

- **Short LOC:** In some cases, the detected clones correspond to a very few LOC. In these cases, extracting and generalizing the code into a reusable unit may result in the introduction of more lines than are removed. We observed that CPD could even report incomplete fragments of code as clones. These fragments could not be refactored because they lacked the necessary context to form a coherent reusable unit.

- **Comment Lines:** Clone detection tools also report comment lines as part of the detected clones in some cases. This issue was particularly evident in the *Toga project*, where 4 clones identified by Simian were primarily composed of comment lines.

Clones reported by Simian were generally large code Blocks that could be refactored by extracting the repeated logic into helper functions or classes. This process led to a reduction in duplicated code and improved the maintainability of the test code. However, some clones could not be refactored due to comment lines or dependencies on other parts of the code. CPD operates on a token-based detection method, which often resulted in smaller or fragmented code Blocks being identified as clones. This made it difficult to refactor in some cases, as the identified clones lacked the necessary context. In contrast

Table 6.1: *Refactorability of testing code based on detected clones across projects.*

Project	Tool	Granularity	Refactorable Clone Class	Unrefactorable Clone Class				Rate
				Code Dep.	Lack of Sim.	Short LOC	Comm. Lines	
Home Assistant	Simian	Block*	85	8	0	0	7	85%
	CPD	Block*	80	1	0	16	4	80%
	NiCad	Block*	63	12	9	16	0	63%
		File	15	3	3	9	0	50%
		Function*	73	9	4	14	0	73%
Kivy	Simian	Block*	40	42	0	15	3	40%
	CPD	Block*	30	54	0	7	9	30%
	NiCad	Block	15	33	4	5	0	26%
		File	0	0	0	0	0	0%
		Function	13	39	1	2	0	24%
Toga	Simian	Block*	93	3	0	0	4	93%
	CPD	Block*	91	3	0	6	0	91%
	NiCad	Block	71	5	2	7	0	84%
		File	4	0	0	0	0	100%
		Function	81	4	4	11	0	81%
Smart TV	Simian	Block*	87	0	0	11	2	87%
	CPD	Block*	91	0	0	6	3	91%
	NiCad	Block	28	0	2	4	0	82%
		File	1	0	0	0	0	100%
		Function	21	0	2	3	0	81%

*100 random sample clone classes were taken from the tool output and analyzed.

to Simian and CPD, NiCad provides a similarity rate for the detected clones. Refactoring clones that were reported as 100% identical was mostly straightforward by function extraction. However, even though NiCad marks some clones as 100% identical, it is blind to variable and attribute differences. As a result, variables or attributes that differ between clones must be carefully identified and passed as parameters during refactoring, which increased complexity. For clones with 80%-90% similarity, the differences were more substantial, often involving variations in control flow or logic. Refactoring often led to an increase in the overall LOC. While the cloned sections were generalized, the need for additional parameters and control structures to handle differences sometimes introduced more complexity, resulting in more code than before the refactoring. Clones with less than 80% similarity, the differences were often too significant to apply meaningful refactoring; we could not find enough commonality to justify extracting reusable functions, leading to limited or no refactoring.

6.2 Impact of Refactoring on Code Duplication and Test Code Size (RQ2)

Table 5.1 shows reductions in LOC after automated refactoring. When we check the reductions on a project basis, although the total number of clones refactored was high for the Home Assistant project, and a reduction of 699 lines was achieved, the ratio of this reduction to the total LOC in the project is **0.08%**. When we check this ratio for the Kivy, Toga, and Smart TV projects, we observe a reduction of **0.22%**, **2.61%** and **8.14%**, respectively. Hence, we observe that up to 8% automated reduction in test script size is possible.

In order to estimate the possible reduction in LOC by manual refactoring, we first filtered the Simian and CPD reports to obtain a unique set of reported clones. Then we obtained the estimated reduction in LOC by proportioning the number of cloned lines in these unique clones to the refactorable rates in Table 6.1. In order to reflect this ratio more realistically, we left a margin of error of 10% in each result in order to take into account the lines that may be added due to the refactoring process to be applied (import lines, new function definitions, etc.). Below is a summary of estimations regarding the reduction of LOC for each project:

- **Home Assistant:** The manual refactoring of similar clones found with the Simian tool might lead to a reduction of approximately **225 to 250 Thousands of Line of Codes (KLOC)**. Looking at the same project with the clones found by the CPD tool suggests a possible reduction between **175 and 195 KLOC**.

- **Kivy:** Refactoring clones reported by Simian can result in a reduction between **495 - 550 LOC**. For those found by CPD, a reduction between **560 - 620 LOC** is possible.

- **Toga:** Approximately **3745 to 4160 LOC** reduction can be achieved by addressing

the reports of Simian. The estimation ranges between **3,570 and 3,970 LOC** for CPD.

- **Smart TV:** The refactoring of the clones reported by the Simian tool might lead to a reduction between **2,690 and 2,980 LOC**. The estimation ranges between **3,170 and 3,520 LOC** for the CPD tool.

Overall, the potential for the reduction of LOC differs highly based on the project and its scale. The amount of reduction can range from 0.5 KLOC up to 250 KLOC. In the following, we discuss the impact of refactoring based on two different test types: unit tests and functional tests.

6.3 Impact of Refactoring Performance based on Test Types (RQ3)

We analyzed the effects of refactoring operations on the unit tests and functional testing code that are separately available for some projects. Table 6.2 shows the number of clone classes and clone densities for two types of tests. In order to make a more accurate comparison, files with syntax errors were cleaned as explained in Section 4.1, and clones listed in the output were made unique. Clone densities were calculated as the ratio of the portion of code identified as clones to the entire code (cleaned of syntax errors) of the test type.

Table 6.3 shows the ratio of reduction in the number of clone classes, clone densities, and LOC for unit tests and functional tests as a result of automated refactoring. Although the reduction in LOC is not significant, we observe significant reduction rates for the number of clone classes and clone densities. This shows that the automated refactoring process is successful in the elimination of clones in testing code. The overall reduction of clone classes, densities and LOC in *Home Assistant* and *Kivy* projects is consistently higher for functional tests when compared with unit tests. The exception is the *Toga* project, where clone class reduction is higher for unit tests, but clone density and LOC

Table 6.2: *Number of clone classes and clone densities for unit tests and functional tests.*

Project Name	Tool Name	Unit Test		Functional Test	
		# of Clone Classes	Clone Density	# of Clone Classes	Clone Density
Home Assistant	**Simian**	176	15.4%	213	16.6%
	CPD	69	9.6%	115	13.3%
	NiCad	68	10.9%	37	4.6%
Kivy	**Simian**	89	12.4%	23	15.7%
	CPD	94	19.0%	22	17.5%
	NiCad	38	21.6%	13	8.6%
Toga	**Simian**	101	6.2%	95	14.5%
	CPD	169	16.4%	94	18.2%
	NiCad	51	5.8%	30	18.3%
Smart TV	**Simian**	NA	NA	74	41.7%
	CPD	NA	NA	89	47.0%
	NiCad	NA	NA	49	61.4%

reduction is again higher for functional tests. Overall, we observe that automated refactoring was more effective for refactoring functional tests compared to unit tests. We discuss validity threats for our study in the following.

Table 6.3: *The impact of automated refactoring on the number of clone classes and clone densities for unit tests and functional tests.*

Project Name	Unit Test			Functional Test		
	Clone Class Reduction	Density Reduction	LOC Reduction	Clone Class Reduction	Density Reduction	LOC Reduction
Home Assistant	70.6%	65.0%	0.05%	95.3%	77,5%	0.3%
Kivy	27.8%	39.9%	0.2%	42.9%	44.7%	0.3%
Toga	88.5%	92.0%	1.9%	81.8%	94.4%	5.9%
Smart TV	NA	NA	NA	76.7%	90.8%	8.1%

6.4 Threats to Validity

Our evaluation is subject to an *external validity* threat since the results are highly dependent on the characteristics of the code base. To mitigate this threat, we used three open-source and one industrial project from various domains. Home Assistant, with its extensive and diverse code base, is more likely to have higher duplication due to repetitive patterns in integrating various IoT devices. This expectation was also reflected in the results. In contrast, Kivy and Toga, being more focused on graphical user interface

development, exhibit different patterns of code reuse and duplication.

One potential *internal validity* threat arises from the manual evaluation of refactoring effectiveness. The process of selecting and assessing code clones manually introduces the possibility of bias or inconsistency in the evaluation. Additionally, the effectiveness of automated refactoring may be influenced by specific tool configurations or the expertise of the evaluators.

To assess the effectiveness of refactoring in reducing testing code size, we aimed to investigate both automated and manual refactoring of detected clones. However, a *construct validity* threat exists due to the large number of reported clones for some tools, reaching up to hundreds of thousands of clone pairs in certain projects. This raises concerns about whether the reported clones accurately reflect meaningful duplication or include false positives.

To address the challenge posed by the large number of reported clones, we performed a rough estimation based on a random sample of 100 clones. While this approach provides an initial insight, the limited sample introduces a *conclusion validity* threat. Increasing the sample size could enhance the accuracy of our estimations and strengthen the validity of our conclusions.

7. CONCLUSIONS

We introduce an approach and an integrated set of tools for identifying clones in testing code and refactoring the corresponding code snippets to eliminate duplication and reduce the code size. We first compared the effectiveness of alternative clone detection tools when applied on test scripts developed with Python. We used one of these tools to identify commonalities and code duplication at the function level. Then, we employed static analysis to detect variations in control and data flow in similar functions. In the final step, we applied function generalization or function extraction. Extracted functions are reused by multiple other functions and generalized functions are replaced with multiple similar ones.

We conducted case studies with test suites that include Python scripts developed for testing four different applications. One of these is developed for testing Smart TV systems currently being developed and maintained by a consumer electronics company. The other three are open source projects. Our results obtained with these four experimental objects show that up to 8% reduction in the size of test scripts is possible with automated refactoring. Overall, the potential for the reduction of LOC differs highly based on the project and its scale. The amount of reduction can range from 0.5 KLOC up to 250 KLOC. We also observed that automated refactoring was more effective for refactoring functional tests compared to unit tests in terms of the reduction of the number of clone classes, clone density and LOC.

REFERENCES

[1] B. van Bladel and S. Demeyer, "Clone detection in test code: An empirical evaluation," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 492–500, 2020.

[2] B. Beizer, *Software Testing Techniques*. New York, NY, USA: Van Nostrand Reinhold Co., 2 ed., 1990.

[3] G. Myers, T. Badgett, and C. Sandler, *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley and Sons Inc., 3 ed., 2012.

[4] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering*, pp. 571–579, 2005.

[5] D. Rafi, K. Moses, K. Petersen, and M. Mäntylä, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," in *Proceedings of the 7th International Workshop on Automation of Software Test*, pp. 36–42, 2012.

[6] P. Hamill, *Unit test frameworks: tools for high-quality software development*. " O'Reilly Media, Inc.", 2004.

[7] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Technical debt in test automation," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, pp. 887–892, 2012.

[8] E. Alégroth, M. Steiner, and A. Martini, "Exploring the presence of technical debt in industrial GUI-Based testware: A case study," in *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 257–262, 2016.

[9] E. Alégroth and J. Gonzalez-Huerta, "Towards a mapping of software technical debt onto testware," in *Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications*, pp. 404–411, 2017.

[10] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 56–65, 2012.

[11] D. Lima, R. D. S. Santos, G. Garcia, S. D. Silva, C. França, and L. Capretz, "Software testing and code refactoring: A survey with practitioners," in *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution*, pp. 500–507, 2023.

[12] Q. Ain, W. Butt, M. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, pp. 86121–86144, 2019.

[13] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[14] C. Roy, R. James, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[15] B. van Bladel and S. Demeyer, "A comparative study of test code clones and production code clones," *Journal of Systems and Software*, vol. 176, p. 110940, 2021.

[16] B. van Bladel and S. Demeyer, "A comparative study of code clone genealogies in test code and production code," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 913–920, 2023.

[17] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[18] M. Astekin and H. Sözer, "Utilizing clone detection for domain analysis of simulation systems," in *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 287–291, 2012.

[19] C. Roy and J. Cordy, "Benchmarks for software clone detection: A ten-year retrospective," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 26–37, 2018.

[20] W. Hasanain, Y. Labiche, and S. Eldh, "An analysis of complex industrial test code using clone analysis," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*, pp. 482–489, 2018.

[21] C. Roy and J. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pp. 172–181, 2008.

[22] Z. S. Othman and M. Kaya, "Refactoring code clone detection," in *Proceedings of the 3rd International Conference on Computer Science and Engineering*, pp. 3449–3453, 2018.

[23] F. A. Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering*, vol. 2013, p. 129437, 2013.

[24] N. Tsantalis, D. Mazinanian, and G. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, 2015.

[25] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: Clone refactoring," *In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pp. 613–616, 2016.

[26] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 368–377, 1998.

[27] H. Kaur and R. Maini, "Function clone removal using refactoring techniques," *Advances in Mathematics: Scientific Journal*, vol. 9, no. 6, pp. 4001–4013, 2020.

[28] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, "A novel approach to optimize clone refactoring activity," *Proceedings of the 2006 Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1885–1892, 2006.

[29] M. Badri, L. Badri, O. Hachemane, and A. Ouellet, "Measuring the effect of clone refactoring on the size of unit test cases in object-oriented software: An empirical study," *Innovations in Systems and Software Engineering*, vol. 15, pp. 117–137, 2019.

[30] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: Clone refactoring," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 613–616, 2016.

[31] E. A. AlOmar, B. Knobloch, T. Kain, C. Kalish, M. W. Mkaouer, and A. Ouni, "AntiCopyPaster 2.0: Whitebox just-in-time code duplicates extraction," in *Proceedings of the 46th International Conference on Software Engineering: Companion Proceedings*, pp. 84–87, 2024.

[32] R. Yue, Z. Gao, N. Meng, Y. Xiong, X. Wang, and J. D. Morgenthaler, "Automatic clone recommendation for refactoring based on the present and the past," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 115–126, 2018.

[33] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pp. 18–23, 2016.

[34] B. Zhang, P. Liang, Q. Feng, Y. Fu, and Z. Li, "Copilot-in-the-loop: Fixing code smells in copilot-generated python code using copilot," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, (New York, NY, USA), p. 2230–2234, Association for Computing Machinery, 2024.

[35] Z. Zhang, Z. Xing, X. Xu, and L. Zhu, "Ridiom: Automatically refactoring non-idiomatic python code with pythonic idioms," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 102–106, 2023.

[36] C. S. Gebizli and H. Sözer, "Model-based software product line testing by coupling feature models with hierarchical markov chain usage models," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, pp. 278–283, 2016.

[37] S. Dirim, O. Özener, and H. Sözer, "Prioritization and parallel execution of test cases for certification testing of embedded systems," *Software Quality Journal*, vol. 31, p. 471–496, 2023.

[38] B. V. Bladel and S. Demeyer, "A comparative study of test code clones and production code clones," *Journal of Systems and Software*, vol. 176, p. 110940, 2021.

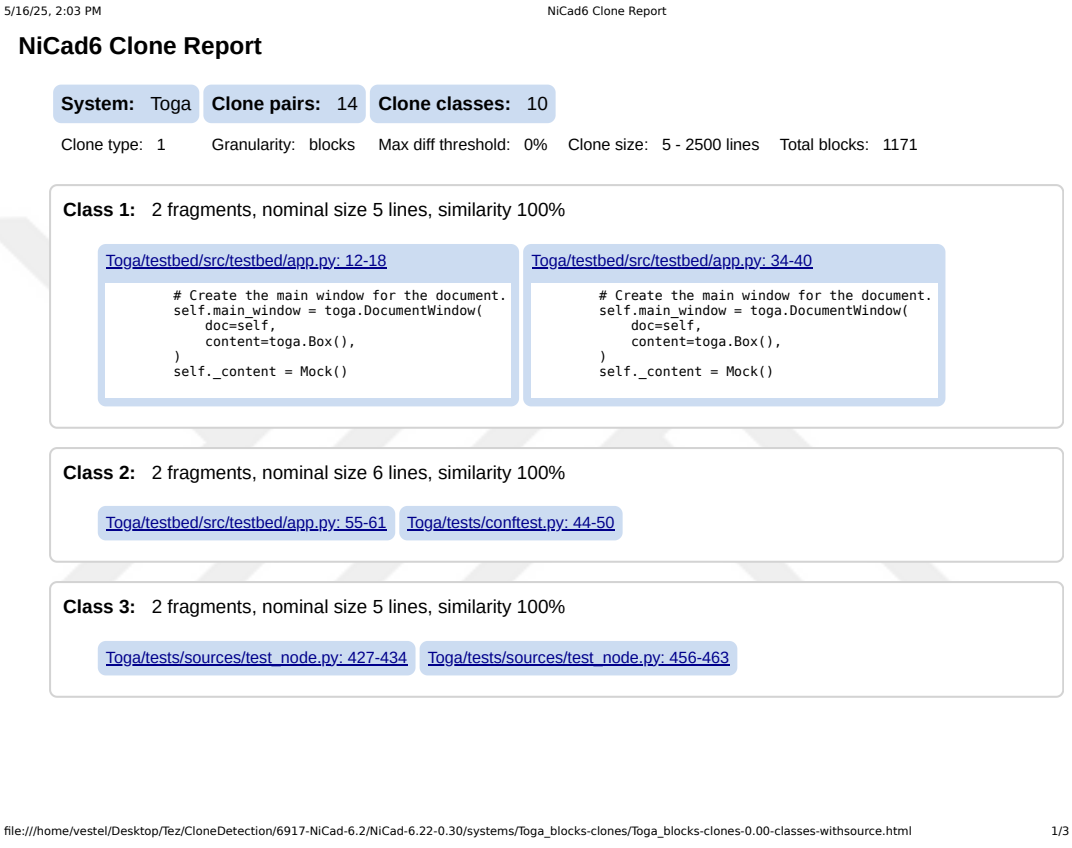
[39] PMD, "Copy/Paste Detector (CPD) 5.0." `https://pmd.sourceforge.io/pmd-5.0.5/cpd-usage.html`, 2024.

[40] Quandary Peak Research, "Tool simian." `https://simian.quandarypeak.com/`, 2024.

APPENDICES

APPENDIX A. SAMPLE OUTPUT OF CLONE DETECTION TOOLS

Figure A.1: *Sample HTML output of clone detection by NiCad on the Toga project*



5/16/25, 2:03 PM NiCad6 Clone Report

NiCad6 Clone Report

System: Toga **Clone pairs:** 14 **Clone classes:** 10

Clone type: 1 Granularity: blocks Max diff threshold: 0% Clone size: 5 - 2500 lines Total blocks: 1171

Class 1: 2 fragments, nominal size 5 lines, similarity 100%

Toga/testbed/src/testbed/app.py: 12-18

```
        # Create the main window for the document.
        self.main_window = toga.DocumentWindow(
            doc=self,
            content=toga.Box(),
        )
        self._content = Mock()
```

Toga/testbed/src/testbed/app.py: 34-40

```
        # Create the main window for the document.
        self.main_window = toga.DocumentWindow(
            doc=self,
            content=toga.Box(),
        )
        self._content = Mock()
```

Class 2: 2 fragments, nominal size 6 lines, similarity 100%

Toga/testbed/src/testbed/app.py: 55-61 Toga/tests/conftest.py: 44-50

Class 3: 2 fragments, nominal size 5 lines, similarity 100%

Toga/tests/sources/test_node.py: 427-434 Toga/tests/sources/test_node.py: 456-463

file:///home/vestel/Desktop/Tez/CloneDetection/6917-NiCad-6.2/NiCad-6.22-0.30/systems/Toga_blocks-clones/Toga_blocks-clones-0.00-classes-withsource.html 1/3

Figure A.2: *Sample XML output of clone detection by Simian on the Toga project*

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="simian.xsl" type="text/xsl"?>
<!--
Simian Similarity Analyzer 4.0.0 - https://simian.quandarypeak.com
Copyright (c) 2023 Quandary Peak Research. All rights reserved.
Subject to the Quandary Peak Academic Software License.
-->
<simian version="4.0.0">
    <check failOnDuplication="true" ignoreCharacterCase="true" ignoreCurlyBraces="true" ignoreIdentifierCase="true" ignoreModifiers="true" ignoreStringCase="true" threshold="10">
        <set lineCount="10" fingerprint="1e2ada371952e00e4106aea5de94ae27">
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests/test_validators.py" startLineNumber="397" endLineNumber="414"/>
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests/test_validators.py" startLineNumber="346" endLineNumber="363"/>
        </set>
        <set lineCount="10" fingerprint="40ef6c7720da3707c34999c13fc4758d">
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests/sources/test_tree_source.py" startLineNumber="229" endLineNumber="238"/>
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests/sources/test_tree_source.py" startLineNumber="166" endLineNumber="175"/>
        </set>
        <set lineCount="10" fingerprint="62b010555d29e0eda350b38540945445">
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests_backend (1)/widgets/base.py" startLineNumber="76" endLineNumber="87"/>
            <block sourceFile="/home/vestel/Desktop/Tez/CloneDetection/simian-academic/simian-4.0.0/TestToga/tests_backend (3)/widgets/base.py" startLineNumber="123" endLineNumber="134"/>
        </set>
```

Figure A.3: *Sample XML output of clone detection by CPD on the Kivy project*

```
2354
2355 ==========================================================================
2356 Found a 13 line (44 tokens) duplication in the following files:
2357 Starting at line 239 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_uix_recyclegridlayout.py
2358 Starting at line 255 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_uix_recyclegridlayout.py
2359
2360             scroll_to=(150, 0), clock=kivy_clock)
2361
2362     # |---|
2363     # |   |
2364     # |---|
2365     # | 1 |
2366     # |---|
2367     # | 2 |
2368     # |---|
2369     # |   |
2370     # |---|
2371     @pytest.mark.parametrize("n_cols, n_rows", [(1, None), (None, 4), (1, 4)])
2372     @pytest.mark.parametrize("orientation", "tb-lr tb-rl lr-tb rl-tb".split())
2373
2374 ==========================================================================
2375 Found a 9 line (43 tokens) duplication in the following files:
2376 Starting at line 437 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/async_common.py
2377 Starting at line 490 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/async_common.py
2378
2379         touch = AsyncUnitTestTouch(x, y)
2380
2381         touch.touch_down()
2382         await self.wait_clock_frames(1)
2383         if long_press:
2384             await self.async_sleep(long_press)
2385         yield 'down', touch.pos
2386
2387         ts0 = time.perf_counter()
2388
2389 ==========================================================================
2390 Found a 7 line (43 tokens) duplication in the following files:
2391 Starting at line 61 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_animations.py
2392 Starting at line 112 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_animations.py
2393 Starting at line 124 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_animations.py
2394 Starting at line 134 of /home/vestel/pmd-bin-7.0.0/Kivy/KivyTest/tests/test_animations.py
2395
2396     def test_start_animation(self):
2397         from kivy.animation import Animation
```

APPENDIX B. SAMPLE UNREFACTORABLE CODE CLONES

Figure B.1: *Sample clone pair detected by NiCad in the Kivy project, considered unrefactorable due to* ***code dependencies***

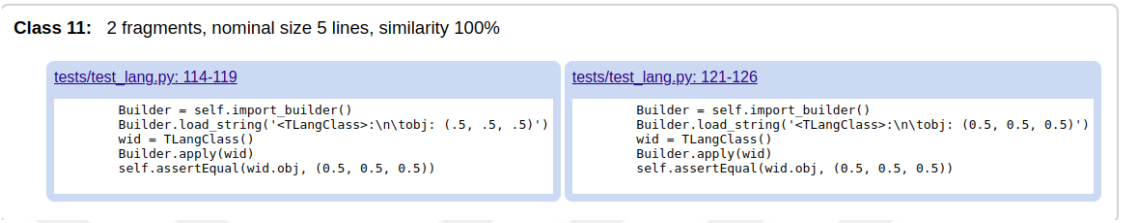


Figure B.2: *Sample clone pair detected by NiCad in the Home Assistant project, considered unrefactorable due to* ***lack of similarity***

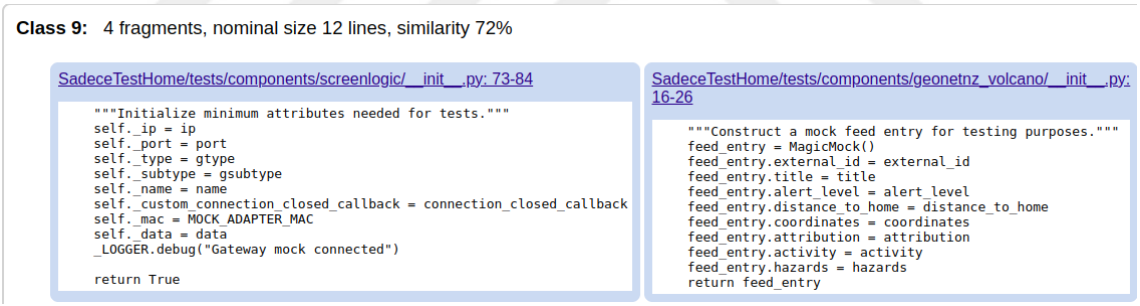


Figure B.3: *Sample clone pair detected by CPD in the Toga project, considered unrefactorable due to* ***short LOC***

```
8752 =====================================================================
8753 Found a 3 line (36 tokens) duplication in the following files:
8754 Starting at line 60 of /home/vestel/pmd-bin-7.0.0/before/TestToga/tests_backend (2)/app.py
8755 Starting at line 119 of /home/vestel/pmd-bin-7.0.0/before/TestToga/tests_backend (5)/app.py
8756
8757                 assert img.getpixel((5, 5)) == (211, 230, 245)
8758                 mid_color = img.getpixel((img.size[0] // 3, img.size[1] // 3))
8759                 assert mid_color == (0, 204, 9)
8760
8761 =====================================================================
8762
8763 Found a 4 line (35 tokens) duplication in the following files:
8764 Starting at line 213 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8765 Starting at line 284 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8766 Starting at line 294 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8767 Starting at line 344 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8768 Starting at line 354 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8769 Starting at line 424 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8770 Starting at line 434 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8771
8772     assert not probe.is_expanded(source[0])
8773     assert not probe.is_expanded(source[0][2])
8774     assert not probe.is_expanded(source[1])
8775     assert not probe.is_expanded(source[1][2])
8776
```

Figure B.4: *Sample clone pair detected by CPD in the Toga project, considered unrefactorable due to* ***comment lines***

```
8506 =====================================================================
8507 Found a 7 line (36 tokens) duplication in the following files:
8508 Starting at line 336 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8509 Starting at line 346 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8510 Starting at line 397 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8511 Starting at line 407 of /home/vestel/pmd-bin-7.0.0/before/TestToga/testbed/tests/widgets/test_tree.py
8512
8513     assert not probe.is_expanded(source[1])
8514     # State of source[1][2] is ambiguous
8515     assert probe.is_expanded(source[2])
8516     assert probe.is_expanded(source[2][2])
8517
8518     # Collapse a single child node
8519     widget.collapse(source[0][2])
8520
8521 =====================================================================
```

APPENDIX C. EXAMPLES OF CODE CLONES DETECTED AND MISSED BY CLONE DETECTION TOOLS

C.1 Clones Detected by Simian but Missed by NiCad

There are code clones left undetected but embodied in large code blocks analyzed by NiCad. For instance, NiCad's block-level clone detection extracts the block depicted in Figure C.1 from the file `test_graphics.py`.

Figure C.1: *A large block extracted by NiCad from* `test_graphics.py`.

```
1
2  <source file="systems/Kivy/tests/test_graphics.py" startline="3" endline="1149">
3  ...
4  INDENT
5      from kivy.uix.widget import Widget
6      from kivy.graphics import Color, SmoothEllipse
7      r = self.render
8      ...
9      wid = Widget()
10     with wid.canvas :
11     INDENT
12         Color(1, 1, 1, 0.5)
13         ellipse = SmoothEllipse(pos = (100, 100), size = (150, 150))
14     DEDENT
15     r(wid)
16     ellipse_center = [
17         ellipse.pos [0] + ellipse.size [0] / 2,
18         ellipse.pos [1] + ellipse.size [1] / 2,
19         ]
20     filtered_points = self._filtered_points(
21         ellipse.points + ellipse_center)
22     assert (
23         ellipse.antialiasing_line_points
24         == filtered_points + filtered_points [: 2])
25     ...
26 DEDENT
27 ...
28 </source>
```

As seen in Figure C.1, NiCad captures a wide block (lines 3–25) under its `block` granularity setting. Repeated code within this block is not separated as individual clone candidates. Despite being visually identical and repeated within the same function body, the code block depicted in Figure C.2 was not extracted as a separate block by NiCad, and therefore not considered for clone detection.

Figure C.2: *Clone pair detected by Simian but missed by NiCad*

```
"""Kivy/tests/test_mouse_multitouchsim.
    py: 793-804"""

r(wid)
ellipse_center = [
    ellipse.pos[0] + ellipse.size[0] /
    2,
    ellipse.pos[1] + ellipse.size[1] /
    2,
]
filtered_points = self._filtered_points
    (
    ellipse.points + ellipse_center
)
assert (
    ellipse.antialiasing_line_points
    == filtered_points +
    filtered_points[:2]
)

```

```
"""Kivy/tests/test_mouse_multitouchsim.
    py: 843-854"""

r(wid)
ellipse_center = [
    ellipse.pos[0] + ellipse.size[0] /
    2,
    ellipse.pos[1] + ellipse.size[1] /
    2,
]
filtered_points = self._filtered_points
    (
    ellipse.points + ellipse_center
)
assert (
    ellipse.antialiasing_line_points
    == filtered_points +
    filtered_points[:2]
)

```

The block shown in Figure C.2 is located twice within the same function in `test_graphics.py`. This code block corresponds to lines 15 to 22 within the code block depicted in Figure C.1; however, NiCad misses this portion because it does not analyze the contents within the block. Simian, which operates on textual similarity and sliding window comparison, successfully detects this duplicated fragment. However, NiCad fails to identify it due to its block-level granularity, where both instances reside inside a single large block and are therefore not extracted as separate clone candidates. This demonstrates a fundamental limitation of NiCad's block granularity when repeated structures are embedded within a single control flow region.

C.2 Clones Detected by NiCad but Missed by Simian

Figure C.3 shows a clone pair detected exclusively by NiCad between two test files in the `Toga` project. Despite their textual and structural similarity, the Simian tool did not report them as clones.

Figure C.3: *Clone pair detected only by NiCad*

```
"""Toga/tests/widgets/
    test_multilinetextinput.py: 90-106
    """

"""The readonly status of the widget
    can be changed."""
# Widget is initially not readonly by
    default.
assert not widget.readonly

# Set the readonly status
widget.readonly = value
assert widget.readonly == expected

# Set the widget readonly
widget.readonly = True
assert widget.readonly

# Set the readonly status again
widget.readonly = value
assert widget.readonly == expected

```

```
"""Toga/tests/widgets/test_textinput.py
    : 158-174"""

"""The readonly status of the widget
    can be changed."""
# Widget is initially not readonly by
    default.
assert not widget.readonly

# Set the readonly status
widget.readonly = value
assert widget.readonly == expected

# Set the widget readonly
widget.readonly = True
assert widget.readonly

# Set the readonly status again
widget.readonly = value
assert widget.readonly == expected

```

Upon re-examining the results of Simian, we observed that it identifies a larger block encompassing the clone shown in Figure C.4. The code block in Figure C.3 corresponds to lines 16 to 29 of Figure C.4. To understand the reason behind this, we also checked NiCad's XML output and found, as illustrated in Figure C.5, that NiCad splits this region into a smaller block. Therefore, while NiCad detects only a smaller portion of the clone in the same location, Simian identifies a larger segment, which leads to two separate clone detections by the tools.

Figure C.4: *Simian clone block encompassing NiCad-detected clone*

```
"""Toga/tests/widgets/
    test_multilinetextinput.py: 78-113
    """

@pytest.mark.parametrize(
    "value, expected",
    [
        (None, False),
        ("", False),
        ("true", True),
        ("false", True),  # Evaluated
    as a string, this value is true.
        (0, False),
        (1234, True),
    ],
)
def test_readonly(widget, value,
    expected):
    """The readonly status of the
    widget can be changed."""
    # Widget is initially not readonly
    by default.
    assert not widget.readonly

    # Set the readonly status
    widget.readonly = value
    assert widget.readonly == expected

    # Set the widget readonly
    widget.readonly = True
    assert widget.readonly

    # Set the readonly status again
    widget.readonly = value
    assert widget.readonly == expected


@pytest.mark.parametrize(
    "value, expected",
    [
        ("New Text", "New Text"),
        ("", ""),
        (None, ""),
        (12345, "12345"),
```

```

"""Toga/tests/widgets/test_textinput.py
    : 146-181"""

@pytest.mark.parametrize(
    "value, expected",
    [
        (None, False),
        ("", False),
        ("true", True),
        ("false", True),  # Evaluated
    as a string, this value is true.
        (0, False),
        (1234, True),
    ],
)
def test_readonly(widget, value,
    expected):
    """The readonly status of the
    widget can be changed."""
    # Widget is initially not readonly
    by default.
    assert not widget.readonly

    # Set the readonly status
    widget.readonly = value
    assert widget.readonly == expected

    # Set the widget readonly
    widget.readonly = True
    assert widget.readonly

    # Set the readonly status again
    widget.readonly = value
    assert widget.readonly == expected


@pytest.mark.parametrize(
    "value, expected",
    [
        ("New Text", "New Text"),
        ("", ""),
        (None, ""),
        (12345, "12345"),
```

According to the questions.txt file included in the NiCad distribution and Cordy's work [21], In block granularity mode, NiCad detects clones within compound statements, such as structured control constructs (if, while, for, etc.) or begin-end blocks of a certain minimum size. In contrast, Simian ignores structured block boundaries during parsing, which can lead to the detection of larger or smaller clone blocks.

Figure C.5: *Block extracted by NiCad from* `test_multilinetextinput.py`

```
...
</source>
<source file="systems/Toga/tests/widgets/test_multilinetextinput.py" startline="90"
    endline="106">
INDENT
    assert not widget.readonly
    widget.readonly = value
    assert widget.readonly == expected
    widget.readonly = True
    assert widget.readonly
    widget.readonly = value
    assert widget.readonly == expected
DEDENT
</source>
<source file="systems/Toga/tests/widgets/test_multilinetextinput.py" startline="118"
    endline="131">
INDENT
    EventLog.reset()
    widget.placeholder = value
    assert widget.placeholder == expected
    assert attribute_value(widget, "placeholder") == expected
    assert_action_performed(widget, "refresh")

DEDENT
</source>
<source file="systems/Toga/tests/widgets/test_multilinetextinput.py" startline="133"
    endline="150">
INDENT
    EventLog.reset()
    widget.scroll_to_top()

    assert_action_performed(widget, "scroll to top")

    EventLog.reset()
    widget.scroll_to_bottom()

    assert_action_performed(widget, "scroll to bottom")

DEDENT
...
```

C.3 Commonly Detected Clones

Figure C.6 shows a code fragment that was reported as a clone by both tools (NiCad and Simian). This example involves a structurally and textually identical Kivy test code in the same file but at different locations.

Figure C.6: *Clone pair detected by both tools (NiCad and Simian)*

```
"""Kivy/tests/test_lang.py: 198-213"""

Builder = self.import_builder()
Builder.load_string(dedent('''
<TLangClass>:
    on_press:
        print('hello world')
        print('this is working !')
        self.a = 1
'''))
wid = TLangClass()
Builder.apply(wid)
wid.a = 0
self.assertTrue('on_press' in wid.
    binded_func)
wid.binded_func['on_press']()
self.assertEqual(wid.a, 1)
```

```
"""Kivy/tests/test_lang.py: 232-247"""

Builder = self.import_builder()
Builder.load_string(dedent('''
<TLangClass>:
 on_press:
  print('hello world')
  print('this is working !')
  self.a = 1
'''))
wid = TLangClass()
Builder.apply(wid)
wid.a = 0
self.assertTrue('on_press' in wid.
    binded_func)
wid.binded_func['on_press']()
self.assertEqual(wid.a, 1)
```