View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Test Clone Detection via Assertion Fingerprints

by

Zheng (Felix) Fang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Computer Software

Waterloo, Ontario, Canada, 2014

© Zheng (Felix) Fang 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Large software systems require large test suites to achieve high coverage. Test suites often employ closed unit tests that are self-contained and have no input parameters. To achieve acceptable coverage with self-contained unit tests, developers often clone existing tests and reproduce both boilerplate and essential environment setup code as well as assertions. Existing technologies such as parametrized unit tests and theories could mitigate cloning in test suites. These technologies give developers new ways to express refactorings. However, they do not help detect refactorable clones in the first place, which requires tedious manual effort.

This thesis proposes a novel technique, *assertion fingerprints*, for detecting clones based on the set of assertion/fail calls in test methods. Assertion fingerprints encode the control flow around the ordered set of assertions in methods.

We have implemented clone set detection using assertion fingerprints and applied it to 10 test suites for open-source Java programs. We provide an empirical study and a qualitative analysis of our results. Assertion fingerprints enable the discovery of test clones that exhibit strong structural similarities and are amenable to refactoring. Our technique delivers an overall 75% true positive rate on our benchmarks and identifies 44% of the benchmark test methods as clones.

Acknowledgements

I would like to thank Patrick Lam's invaluable advice and guidance which made this thesis possible. I would also like to thank Divam Jain for his prior work on test clone detection in his Master's thesis and Wei Wang's assistance in discussion of the Bauhaus clone detector.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 Approach 4

2.1 Clone Definitions 4

2.2 Motivating Example 6

2.3 Assertion Fingerprints 6

2.4 Building Clone Sets 11

3 Results 14

3.1 Empirical Study 14

3.2 Qualitative Analysis 18

3.3 Large Clone Sets 24

4 Discussion 27

4.1 False and Fragmented True Positives 27

4.2 Refactorability of Clones 28

4.3 Limitations 29

4.4 Comparison with Clone Detectors 29

4.5 Threats to Validity 31
4.6 Future Work 33

5 Related Work **37**

6 Conclusions **39**

APPENDICES **40**

A Output Sample **41**

References **43**

List of Tables

3.1 Statistics of detected clones plaguing 44% methods in test suites and showing clone sets tend to have few methods (3), classes (2), and packages (1). . . . 15

3.2 Counts of assertions in clone sets. The median clone set contains 4 assertions. 17

3.3 Sampling-based investigation shows that 75% of the reported clone sets are true positives. Confidence intervals included for 95% confidence level. . . . 18

3.4 Filtering is effective—87% of filtered-out clone sets are either false positives or fragmented true positives. 19

4.1 How Bauhaus performed against Assertion Fingerprints—45% of clone sets from Assertion Fingerprints were not detected by Bauhaus. 31

4.2 How Assertion Fingerprints performed against Bauhaus—Assertion Fingerprints found 49% of clone sets detected by Bauhaus. 32

List of Figures

2.1 A clone set of four refactorable test method clones reported from Weka's test suite (`weka.filters.unsupervised.attribute.RemoveTypeTest`). . . 7

2.2 A control-flow graph for the example in Figure 2.1. Only assertions and predicates included. 8

2.3 Refactored `testFiltering()` motivating example from Figure 2.1 (using standard Java). 9

2.4 A example illustrating assertion fingerprints. 11

2.5 A control-flow graph over three-address code for the example in Figure 2.4. We elide all but assertions and predicates. Notations `bc`, `mc`, `es`, `inLoop`, `inCatch` represent branch count, merge count, exceptional successors, whether in loops or not, and whether in a catch block or not. 12

3.1 Distribution of clone set sizes. Each point represents one clone set, showing the number of asserts common to members of that clone set on the x-axis, and its distribution (# of similar methods) on the y-axis. 98.26% of clone sets have fewer than 40 assertions and 50 methods. 16

3.2 A clone set from the JGraphT test suite (`org.jgrapht.experimental.GraphReaderTest`). Details are elided. 19

3.3 JDOM uses try/catch/`fail()` to ensure certain exceptions are thrown. . . 20

3.4 Two implementations of `testEquality()` make up one clone set from the Joda-Time test suite. Both contain straight-line assertion fingerprints. . . 21

3.5 Test method from the Apache Commons Collections, belonging to classes `Test{ArrayIterator, IteratorChain, ObjectArrayIterator, UniqueFilterIterator}`. At runtime, `makeFullIterator()` (line 2) and the value of `testArray` (line 3, 4) bind different values depending on the containing class. 22

3.6 JFreeChart contains `testClear()` methods which are identical except for the type under test. 24

3.7 A false positive clone set from the Apache POI's test suite. Not considering the arguments passed to the assertions causes this false positive. 25

3.8 A control-flow graph for method `testCloning()` in the JFreeChart test suite. Irrelevant details elided: we only include assertions and predicates. 26

4.1 Four test methods of a fragmented true positive clone set from the Apache Commons Collections test suite. Despite all having the same assertion fingerprint, only the first two methods are clones, and the last two methods are clones. 34

4.2 A non-parametrizable clone set from the Apache Commons Collections test suite. 35

4.3 A member of an ubiquitous clone set from Weka. Identical copies of `testToolTips()` exist in 10 test classes. 35

4.4 A JDOM test method using a helper method. Helper method assertions are invisible to our technique. 36

Chapter 1

Introduction

Modern software systems often use unit tests to ensure proper code behaviour. Ideally, a suite of unit tests works together to cover the system's functionality. However, achieving acceptable coverage using popular unit testing frameworks, such as JUnit, often requires the use of repetitive boilerplate code. The problem is that traditional unit test design recommends self-contained test classes—each test class must run independently and take no input parameters. The easiest way to create such self-contained test classes is to clone boilerplate test code, which increases test maintenance overhead and propagates any pre-existing errors.

Our goal is to facilitate the refactoring of suites of unit tests by detecting test clones. In this thesis, we describe the design and implementation of a tool that identifies cloned unit tests, using assertion fingerprints, and provides contextual information about these clones to the developer.

A number of technologies already exist for implementing test refactoring. Tillman and Schulte have proposed *parametrized unit tests (PUTs)* to increase the expressive power of unit testing and to enable test reuse [12]. Similarly, Saff proposed *theories* to simplify and increase the robustness of unit tests [7]. Developers may also use language features such as inheritance or generics to refactor tests.

We believe that retrofitting existing test suites to reduce undesirable clones is valuable. According to Saff, the use of theories reduces the long term maintenance cost for test suites [10]. Moreover, Thummalapenta et al. conclude from an empirical study of existing test suites that parametrization is beneficial—their results indicate that test suites, when retrofitted with parametrization, can detect new defects and provide increased branch

coverage [11]. Test refactoring also generally reduces the brittleness and improves the ease-of-understanding of test code.

However, test refactoring technologies currently require the developer to manually identify opportunities for refactoring, which becomes increasingly difficult as test suite size continues to grow. While writing new tests using test refactoring techniques is often the correct long-term decision, it is not always clear when such techniques apply. Incremental evolution and short-term imperatives may slowly lead to situations where test cloning gets out of control. The output of our tool—sets of likely test clones—enables developers to regain control, refactor test clones, and improve the quality of their test suites.

There is a rich body of work on clone detection. However, existing clone detection techniques usually treat the code as a sequence of input tokens; they generally do not have a deeper understanding of the code under analysis and can often be foiled by, for instance, changes to the order in which a test computes assertion parameters. Our work leverages insights into the structure of unit tests to identify clones and to give more useful feedback to developers about where clones exist, enabling targeted refactoring efforts. Refactoring yields a test suite with fewer clones. Simpler suites are easier to keep up-to-date as the system under test evolves.

We have formulated a technique for analyzing suites of unit tests and detecting clones. We implemented our technique using the Soot program analysis framework [8]. Our tool analyzes test suites and displays likely-cloned sets of test methods along with the evidence, in terms of specific program fragments, used to draw this inference.

Using our tool, we conducted an empirical study based on a suite of 10 benchmark programs (ranging from 8,000 to 246,000 lines of code) with test suites ranging from 6,000 to 57,000 lines of test code, and 12,332 test methods in all. Our technique finds that the benchmarks have from 19% to 70% of cloned test methods, with a suite-wide average of 44% clones. We manually inspected 191 randomly sampled clones from these tests and found that 75% of our recommendations were indeed true positive clones.

Our primary contributions are:

- a technique for identifying test-level clones in existing test suites based on assertion fingerprints;
- an implementation of that technique; and,
- an empirical study of a significant suite of benchmark programs using our technique.

Our technique computes, for each test method, an ordered set of fingerprinted assertions. It then identifies sets of tests with identical ordered sets of assertions, and filters out likely false positives. Finally, our display tool shows the computed clone sets to the developer. In our experience, our tool reports many refactorable clones and few false positives.

We also compared our results with those from the best-in-class clone detector Bauhaus [5]. Our results provided more semantically-similar clones and more insights on their similarities, compared with Bauhaus's texually similar clones.

Our tool is available under the GNU GPL at:

https://bitbucket.org/felixfangzh/cloneanalysis.

Chapter 2

Approach

Our technique detects clones among methods in a test suite by collecting and matching sets of assertions across methods. An assertion fingerprint is an ordered set of assertions, where each assertion has an associated fingerprint consisting of control-flow information. We compute assertion fingerprints, collect sets of methods with matching fingerprints, and filter out common false positive patterns.

We will first give the definition of clones in our context, continue with a motivating example, and then describe how our technique works.

2.1 Clone Definitions

Our analysis aims to detect function clones (Definition 2) in a test suite. We use terms *function clones* and *method clones* interchangeably in this thesis.

Because our technique operates on Java bytecode, it is oblivious to comments and syntactic changes in the code. By design, it works well on semantic clones (Type III, Definition 5), and our approximation and filtering techniques (Table 3.4) help reduce the false positive rate.

The following definitions are adapted from Roy and Cordy's survey on clone detection [9].

Definition 1 *Structural clones are simple clones within a syntactic boundary following the syntactic structure of a particular language. A structural clone can be any of the four types of simple clones (Type I, II, III, and IV) based on its similarity level.*

Definition 2 *Function clones are structural clones where the syntactic boundary is a method boundary.*

Definition 3 *Type I clones are identical code fragments except for variations in whitespace (may be also variations in layout) and comments.*

Definition 4 *Type II clones are structurally/syntactically identical fragments except for variations in whitespace, identifiers, literals, types, layout and comments.*

Definition 5 *Type III clones are copied fragments with further modifications. Statements can be changed, added, or removed in addition to variations in identifiers, literals, types, layout and comments.*

Definition 6 *Type IV clones are two or more code fragments that perform the same computation but implemented through different syntactic variants.*

By construction, all methods obey our definition of a clone set and are hence similar. Some members of clone sets are obviously cut-and-pasted clones, while others are obviously false positives, not clones. (Consider methods with just a single `assertTrue()` call; our technique explicitly filters out such methods, as described in Section 2.4, but they would otherwise be included in the results.) We used our best judgment to classify our results. We classify the results as true positives; false positives; or fragmented true positives.

Definition 7 *A clone set C is a true positive if every method $m \in C$ is a clone of method m' for all $m' \in \{C \setminus m\}$.*

Definition 8 *A clone set C is a false positive if there does not exist a method $m \in C$ that is a clone of method m' for all $m' \in \{C \setminus m\}$.*

Definition 9 *A clone set C is a fragmented true positive if it is not a true positive and there exists one method $m \in C$ that is a clone of at least one method $m' \in \{C \setminus m\}$.*

A fragmented true positive, in other words, is not a true positive, but contains at least one pair of cloned methods.

Figure 2.1, 3.2, 3.3, 3.4, 3.5, 3.6 exemplify true positives, whereas the clone set in Figure 3.7 is a false positive. Figure 4.1 gives an example of a fragmented true positive.

Note that a method can belong to at most one clone set.

2.2 Motivating Example

The goal of our analysis is to report clone sets which are candidates for refactoring by the developer. Our primary assumption is that assertions are frequently a key part of test methods. Therefore, assertion characteristics help us understand the structure and semantics of test methods, and we can use that understanding to detect clones.

Intuitively, our analysis aims to declare that m_1 is a clone of method m_2 if they are similar in terms of control flow and assertion/fail calls.

Consider, as an example, Figure 2.1, from a clone set reported by our analysis. The full set contains 4 similar methods from Weka's test suite. Manual inspection confirms that these methods are indeed clones. Figure 2.3 shows one plausible refactoring of the code from Figure 2.1. While, after refactoring, individual tests become more complicated to understand, we argue that the suite of 4 refactored tests is collectively easier to understand and maintain.

Our analysis identifies the clone set in Figure 2.1 by matching fingerprints from the assertion invocations in the methods. In this case, all four test methods only have one assertion invocation, `assertTrue(boolean)`.

We next compute assertion fingerprints for each of the assertions. Assertion fingerprints consist of 5 components; we illustrate branch counts and merge counts here. Each assertion is reachable from its method start through a single branch, {`i < result.numAttributes()`}, and thus has a branch count of 1. Furthermore, there is an implicit branch and merge in the boolean comparison in the assertion parameter ({`.type() != ...`}), which leads to a merge count of 1 for each assertion. Finally, these assertions are not in loops, catch blocks, or caught by exception handlers. This gives assertion fingerprints {`(bc:1, mc:1)`} for all asserts.

Because all four of the test methods share the same ordered set of assertion fingerprints, and no other methods have the same fingerprints, our technique reports that the test methods in Figure 2.1 make up a clone set.

2.3 Assertion Fingerprints

Our analysis computes a fingerprint for each JUnit assertion or fail call. It uses the assertion fingerprint, along with the parameter types in the assert calls, to match clone sets.

```
1 public void testNominalFiltering() {
2   m_Filter = getFilter(Attribute.NOMINAL);
3   Instances result = useFilter();
4   for (int i = 0; i < result.numAttributes(); i++)
5     assertTrue(result.attribute(i).type() != Attribute.NOMINAL);
6 }
```

```
1 public void testStringFiltering() {
2   m_Filter = getFilter(Attribute.STRING);
3   Instances result = useFilter();
4   for (int i = 0; i < result.numAttributes(); i++)
5     assertTrue(result.attribute(i).type() != Attribute.STRING);
6 }
```

```
1 public void testNumericFiltering() {
2   m_Filter = getFilter(Attribute.NUMERIC);
3   Instances result = useFilter();
4   for (int i = 0; i < result.numAttributes(); i++)
5     assertTrue(result.attribute(i).type() != Attribute.NUMERIC);
6 }
```

```
1 public void testDateFiltering() {
2   m_Filter = getFilter(Attribute.DATE);
3   Instances result = useFilter();
4   for (int i = 0; i < result.numAttributes(); i++)
5     assertTrue(result.attribute(i).type() != Attribute.DATE);
6 }
```

Figure 2.1: A clone set of four refactorable test method clones reported from Weka's test suite (`weka.filters.unsupervised.attribute.RemoveTypeTest`).

Fingerprints contain 5 components: a branch count; a merge count; an exceptional successors count; and two boolean flags, indicating whether the given assertion is in a loop or a catch block.

Clearly, normal control flow edges, as captured in the branch and merge counts and the loop flag, are important to understanding the control-flow structure of test methods and how assertions make up test methods. We also include exceptional control flow in our

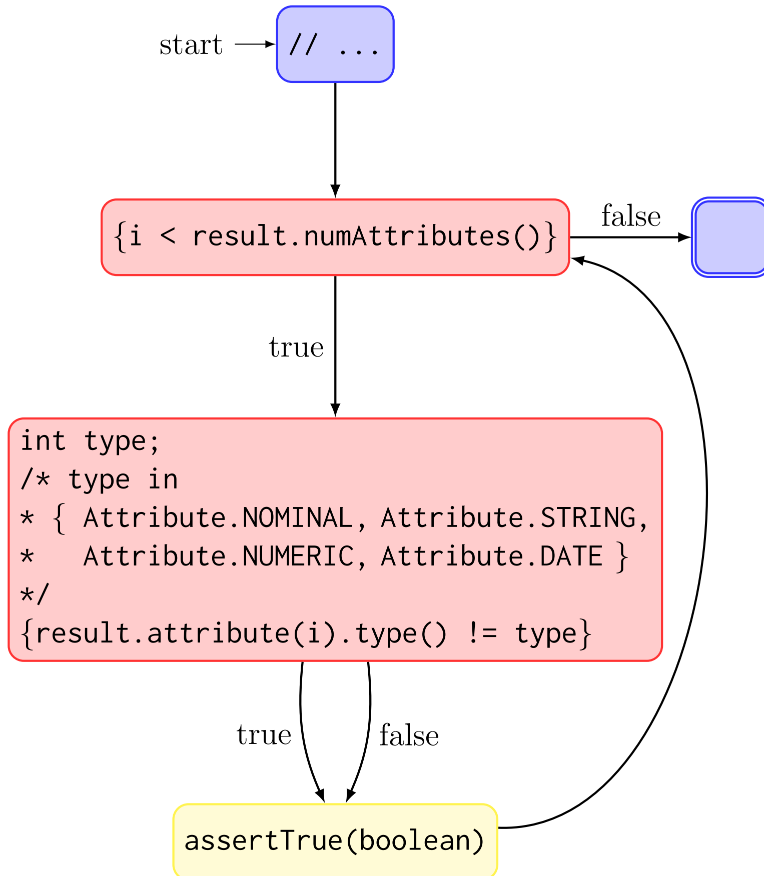

Figure 2.2: A control-flow graph for the example in Figure 2.1. Only assertions and predicates included.

fingerprints after noticing that try and catch blocks are common in some of our test suite benchmarks, particularly JDOM.

Our implementation uses the Soot framework [8] to statically analyze test suites, generate assertion fingerprints, and compute clone sets. Soot provides a control-flow graph (CFG) over three-address code, as well as functions for computing useful properties such as dominators. We have implemented dataflow analyses to compute some of the components of the fingerprints.

Because our technique is only intra-procedural, the implementation can easily exploit multi-core machines.

```
1  static final int [] filteringTypes = {
2    Attribute.NOMINAL, Attribute.STRING,
3    Attribute.NUMERIC, Attribute.DATE
4  };
5
6  public void testFiltering() {
7    for (int type : filteringTypes)
8      testFiltering(type);
9  }
10
11 public void testFiltering(final int type) {
12   m_Filter = getFilter(type);
13   Instances result = useFilter();
14   for (int i = 0; i < result.numAttributes(); i++)
15     assertTrue(result.attribute(i).type() != type);
16 }
```

Figure 2.3: Refactored `testFiltering()` motivating example from Figure 2.1 (using standard Java).

Branch Count We characterize assertions by the control-flow necessary to reach them.

Definition 10 *The branch count of a statement is the minimal number of branches needed to reach that statement from the start of a method.*

We traverse the control-flow graph from its head, annotating each node with a branch count. Every time our traversal encounters a branch, we increment the branch count.

Merge Count The merge count is similar to the branch count, except that the definition of a merge is more complicated than that of a branch. Following each branch vertex b, we annotate successors of b with a pair consisting of the predicate of b and a boolean polarity value—{`true`} for the true branch and {`false`} for the false branch. We ignore nodes with more than 2 successors or predecessors.

Definition 11 *A merge is a vertex m in the control-flow graph with two predecessors, such that the shortest paths from the entry node to m's predecessor vertices p include the*

same set of predicates at branch vertices, and exactly one pair of predicates have opposite polarities.

Definition 12 *The merge count of a statement is the minimal number of merges needed to reach that statement from the start of a method.*

Following a merge, we remove the two predicates with opposite polarities from the annotation sets.

Exceptional Successors We determine the number of exceptional successors of an assertion by examining the exceptional CFG of the method body. Soot's exceptional CFG augments the regular CFG with exception edges, representing possible exceptions and their handling. We consider only the number of exceptional successors and not the types of exceptions being handled.

Loops & Catch Block Indicator Flags We use a depth-first search on the CFG to determine whether an assertion is in a loop. While we do not explicitly record loop depths, a higher branch count is suggestive of more deeply-nested loops.

We determine whether an assertion is in a catch block by using dominator analysis and examining whether any exception handler statement is a dominator of the assertion call in the CFG.

On approximation Our technique collects fingerprints for the assertions in each test method and then identifies methods containing the same assertion fingerprints as potential clones. It would also be possible to use graph isomorphism on control-flow graphs (or slices thereof). However, our fingerprinting technique incorporates approximation, thus capturing more clones than a strict isomorphism-based approach, while empirically showing a low false positive rate. (Section 3 discusses our estimate of the false positive rate.)

Example The code in Figure 2.4, and its CFG in Figure 2.5, illustrate all five components of a fingerprint. For expository reasons, we have included fingerprints for each vertex v in Figure 2.5 (even though only fingerprints for assertion invocations matter for our technique).

Observe that the branch vertex at line 3 ({`i < 10`}) increments the branch count for its successors at line 4 and 5 (inside the for loop) and those at line 10 and 12 (outside

the for loop). Line 6 has two merges: one from {`i == 2`} and the other from {`i != 10`}. Although line 10 (`fail(String)`) is actually unreachable and hence does not throw any exceptions, Soot includes its exceptional successors anyway.

```
1  public void test() {
2     int i;
3     for (i = 0; i < 10; ++i) {
4        if (i == 2)
5           assertEquals(i, 2);
6        assertTrue(i != 10);
7     }
8     try {
9        throw new Exception();
10       fail("Should have thrown exception");
11    } catch (final Exception e) {
12       assertEquals(i, 10);
13    }
14 }
```

Figure 2.4: A example illustrating assertion fingerprints.

2.4 Building Clone Sets

After computing ordered sets of assertion fingerprints, we then partition a suite's test methods into clone sets. Clone sets are sets of methods with the exact same ordered set of assertion fingerprints. We match methods based on the order of assertions because our fundamental assumption is that test methods that verify state (using asserts) in the same order are likely clones.

The final output is displayed in the fashion shown in Appendix A.

Filtering To mitigate the false positive rate, we filter out clone sets unlikely to represent actual test clones. Through manual analysis of the results, we have discovered that the ordered set of assertions of a true positive clone set generally satisfy at least one of:

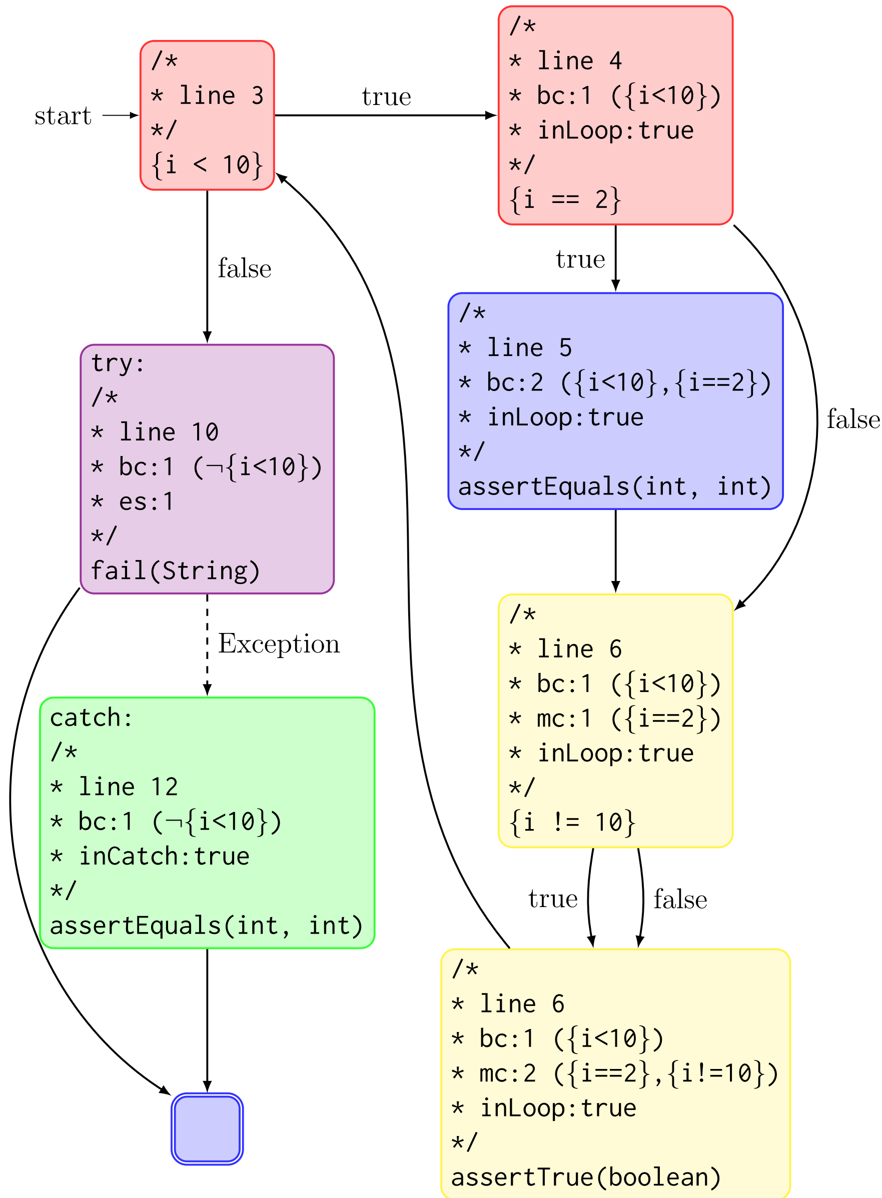

Figure 2.5: A control-flow graph over three-address code for the example in Figure 2.4. We elide all but assertions and predicates. Notations `bc`, `mc`, `es`, `inLoop`, `inCatch` represent branch count, merge count, exceptional successors, whether in loops or not, and whether in a catch block or not.

1. must contain some control flow—this is enforced by requiring that at least one component of an assertion fingerprint is either greater than 0 or true;
2. must contain more than 4 assertions; or
3. must be heterogeneous in signature (invoke different assertion types, e.g. `assertEquals(String, double, double, double)` and `assertEquals(int, int)`).

We only report clone sets which satisfy at least one of the above conditions.

This filtering eliminates clone sets containing unrelated tests. Condition 1 eliminates tests that are too simple; Condition 2 guarantees that test methods are large enough to be interesting; Finally, Condition 3, when the assertion calls do not satisfy conditions 1 or 2, empirically helps avoid false positives on simple (homogeneous) but long test methods.

Chapter 3

Results

We applied our technique to our benchmark suite, which consists of 10 open-source benchmarks. We present an empirical study of the results, a qualitative analysis of our technique's efficacy, and a sampling-based investigation of the false-positive rate.

Our results indicate that cloning is ubiquitous in tests and that our technique successfully identifies 44% of the methods in the suite as being a clone of some other method. Our technique also enjoys a low false positive rate; 75% of the results reported by our technique appear to be similar enough to be refactorable (although perhaps not using standard Java 8).

3.1 Empirical Study

Table 3.1 presents statistical properties of our detected clone sets including counts of methods, classes, and packages per set. It also presents medians, means, and (assuming a normal distribution) standard deviations (σ) for these counts. Clone sets are small and local: the median number of methods in a clone set is 3, classes 2, and packages 1. The skewness of the counts of methods (9.15), classes (14.75), and packages (4.70) per clone set indicate that the counts are positively skewed and tend to be lower than the mean.

Test Methods Table 3.1 shows that 5476 test methods (44% of all test methods in 10 test suites) belong to clone sets, reflecting the severity of cloning in test suites. However, the mean masks a wide dispersion; clone sets contain from 2 to 201 methods, and the suite-wide standard deviation in the number of methods per clone set is 12.3. Joda-Time

Table 3.1: Statistics of detected clones plaguing 44% methods in test suites and showing clone sets tend to have few methods (3), classes (2), and packages (1).

	Clone Sets	Methods	% Methods in Clone Sets	Methods/Set			Classes	Classes/Set			Packages	Packages/Set		
				Median	Mean	σ		Median	Mean	σ		Median	Mean	σ
Apache POI	185	747	26	2	4.0	5.1	571	2	3.1	4.4	411	2	2.2	1.8
Commons Collections	166	501	46	2	3.0	2.2	354	2	2.1	1.8	267	1	1.6	1.0
Google Visualization	35	142	37	3	4.1	3.8	83	2	2.4	1.4	54	1	1.5	0.8
HSQLDB	65	298	40	2	4.6	8.5	99	1	1.5	1.8	75	1	1.2	0.5
JDOM	29	82	31	2	2.8	1.7	37	1	1.3	0.7	32	1	1.1	0.3
JFreeChart	146	1066	49	3	7.3	16.6	900	2	6.2	16.3	373	2	2.6	2.9
JGraphT	11	27	19	2	2.5	0.9	11	1	1.0	0.0	11	1	1.0	0.0
JMeter	60	196	34	2	3.3	2.2	90	1	1.5	1.2	83	1	1.4	1.0
Joda-Time	231	2113	58	4	9.1	19.3	708	2	3.1	3.6	267	1	1.2	0.5
Weka	50	304	70	3	6.1	9.7	139	2	2.8	4.0	100	1	2.0	2.2
Total	978	5476	44	3	5.6	12.3	2992	2	3.1	7.1	1673	1	1.7	1.6

and JFreeChart, with standard deviations of 19.3 and 16.6 in the number of methods per clone set, are particularly widely dispersed.

Distribution of Assertions Since our technique focuses on assertions, we computed statistics about the distribution of assertions in clone sets (Table 3.2). Our results indicate that 44% of assertions in the suite belong to some clone set. Clone sets contain a median of 4 and a mean of 6.2 assertions. Assuming a normal distribution, the standard deviation is 6.6. The skewness (3.54) for assertions is lower than for methods, classes, and packages. This suggests that number of assertions per clone set is comparatively higher than the mean.

Clone Set Size (methods and assertions) Figure 3.1 presents a scatter plot summarizing the sizes (in terms of methods and asserts) of our clone sets. Most of the clone sets reside in the lower left quadrant: 98% of clone sets have fewer than 40 assertions and fewer than 50 methods. Our technique finds zero sets with both lots of methods and lots of assertions.

Estimating True Positive Rate via Sampling Since it is impractical to manually investigate the false positive rate for 978 clone sets, we used random sampling to estimate our false positive rate. For 9 of our benchmarks, we randomly drew 20 reported clone sets and manually classified them as true positives, false positives, or fragmented true positives.

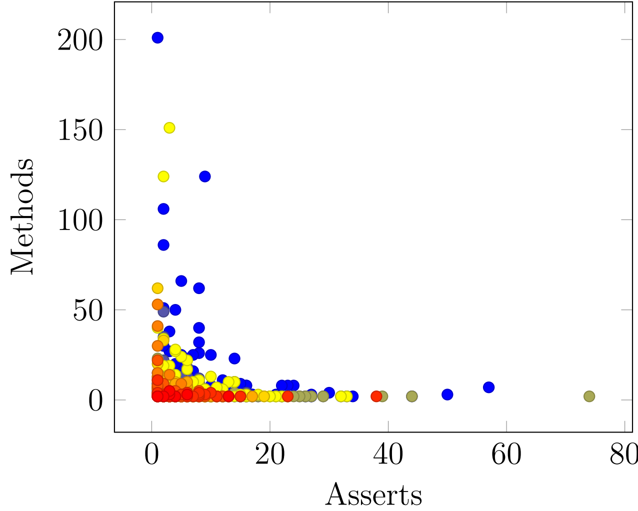

Figure 3.1: Distribution of clone set sizes. Each point represents one clone set, showing the number of asserts common to members of that clone set on the x-axis, and its distribution (# of similar methods) on the y-axis. 98.26% of clone sets have fewer than 40 assertions and 50 methods.

The other benchmark only included 11 clone sets and we manually classified all of them. Table 3.3 shows the manual sampling results. To compute confidence intervals, we assume that the underlying distribution of true/false positive clones is binomial.

The confidence interval of our sampled results is then

$$c_{\frac{\alpha}{2}} \cdot \sqrt{\frac{p(1-p)}{n'}},$$

where $\alpha = 0.05$ is the significance level (giving a confidence level of $1 - \alpha = 95\%$), and p is the probability that a clone set is a true positive. n' is the adjusted sample size for infinite population, and is defined as:

$$n' = \begin{cases} \frac{n \cdot (P-1)}{P-n} & \text{if } P > n; \\ P & \text{if } P \leq n. \end{cases}$$

n is the predetermined sample size, but we use n' to calculate confidence intervals because the statistic model for confidence interval is designed for an infinite population. P is the population size (in this context, the number of clone sets). $c_{\frac{\alpha}{2}}$ is the critical value; we

Table 3.2: Counts of assertions in clone sets. The median clone set contains 4 assertions.

	Clone Sets	Assertions	% Assertions in Clone Sets	Assertions/Set Median	Mean	σ
Apache POI	185	906	17	3	4.9	4.8
Commons Collections	166	1217	46	5	7.3	8.8
Google Visualization	35	147	23	4	4.2	2.8
HSQLDB	65	213	37	2	3.3	3.0
JDOM	29	205	31	5	7.1	7.6
JFreeChart	146	1031	56	5	7.1	5.7
JGraphT	11	37	18	2	3.4	3.4
JMeter	60	271	31	3	4.5	3.4
Joda-Time	231	1923	73	6	8.3	7.9
Weka	50	141	60	3	2.8	1.8
Total	978	26952	44	4	6.2	6.6

will use the normal distribution critical value $z_{\frac{\alpha}{2}}$ or the t-distribution critical value $t_{\frac{\alpha}{2}}$ depending on the sample size n:

$$c_{\frac{\alpha}{2}} = \begin{cases} z_{\frac{\alpha}{2}} & \text{if } n \geq 30 \\ t_{\frac{\alpha}{2}} & \text{if } n < 30 \end{cases}$$

Efficacy of Filtering We manually investigated all 197 clone sets that were filtered out. Table 3.4 presents the results. Overall, 24% of the filtered-out results were false positives and 63% were fragmented true positives. Removing these results increased the quality of our reported results. Filtering worked exceptionally well on POI (40% false positives and 58% fragmented true positives on 43 removed sets) but poorly on Weka (43% true positives in 7 sets removed).

Beyond false positives and highly-fragmented true positives, 13% of the results removed by the filter were indeed clones. However, these true positive clones contained small sets of straight-line homogeneous assertions. These clones are therefore likely ubiquitous clones (Subsection 4.2) and would typically be difficult or not worth the effort to refactor. We believe such clone sets provide little value to the developers and should be ignored.

Table 3.3: Sampling-based investigation shows that 75% of the reported clone sets are true positives. Confidence intervals included for 95% confidence level.

	Clone Sets	Sample Size	% True Positives in Samples	% Fragmented True Positives in Samples	% False Positives in Samples	Confidence Interval (%)
Apache POI	185	20	30	15	55	20
Commons Collections	166	20	85	10	5	16
Google Visualization	35	20	60	35	5	15
HSQLDB	65	20	75	15	10	17
JDOM	29	20	85	15	0	9
JFreeChart	146	20	70	15	15	20
JGraphT	11	11	100	0	0	0
JMeter	60	20	85	15	0	14
Joda-Time	231	20	95	5	0	10
Weka	50	20	80	15	5	15
Total	978	191	75	15	10	5

3.2 Qualitative Analysis

We randomly selected 191 clone sets (all 11 clone sets from JGraphT and 20 from each of the other benchmarks) for a manual analysis, including a false positive determination and a qualitative inspection. Using our best judgment, we determined whether each of these clone sets was a false positive. Table 3.3 summarizes per-benchmark false positive rates. We report true/false positive rates as $(r \pm c)$%, where r is the true/false positive rate of the samples, and c is the confidence interval on the reported true/false positive rate. Overall, the samples demonstrated high true positive rate (75 ± 5)% and low false positive rate (10 ± 5)%. Our technique worked particularly well on JGraphT (100% true positives) and Joda-Time (95 ± 10)% true positives, but less well on Google Visualization (60 ± 15)% and Apache POI (30 ± 20)%. The remaining benchmarks showed true positive rates between 70% and 85%.

JGraphT JGraphT is a Java library that implements graph theory objects and algorithms. Our analysis found 11 clone sets in the 54 test classes (14 packages) of the JGraphT test suite (version 0.8.3). All of them are good quality clones that should be refactored. We were surprised by the high quality of JGraphT's clone sets, since the matched assertion fingerprints are fairly simple. For instance, one of our clone sets includes the assertion fingerprint consisting of a single `assertEquals(String, String)` with 1 exceptional successor. (See Figure 3.2.) This clone set contains two similar test methods, `testGraphReader()` and `testGraphReaderWeighted()`, and no other methods. It appears that our technique performs well on JGraphT's test suite because this suite is composed of heterogeneous test

Table 3.4: Filtering is effective—87% of filtered-out clone sets are either false positives or fragmented true positives.

	Clone Sets Removed by Filter	% False Positives Removed	% Fragmented True Positives Removed	% True Positives Removed
Apache POI	43	40	58	2
Commons Collections	28	32	57	11
Google Visualization	13	23	62	15
HSQLDB	15	33	47	20
JDOM	11	36	55	9
JFreeChart	26	15	73	12
JGraphT	9	11	67	22
JMeter	17	18	65	18
Joda-Time	28	0	82	18
Weka	7	14	43	43
Total	197	24	63	13

methods and is fairly small (diminishing the likelihood of getting false positives).

```
1 public void testGraphReader() {
2   GraphReader<Integer, DefaultEdge> reader;
3   try {
4     Graph<Integer, DefaultEdge> g, g2;
5     // ...
6     assertEquals(g2.toString(), g.toString());
7   } catch (IOException e) { }
8 }
```

```
1 public void testGraphReaderWeighted() {
2   try {
3     Graph<Integer, DefaultWeightedEdge> g;
4     WeightedGraph<Integer, DefaultWeightedEdge> g2;
5     // ...
6     assertEquals(g2.toString(), g.toString());
7   } catch (IOException e) { }
8 }
```

Figure 3.2: A clone set from the JGraphT test suite (`org.jgrapht.experimental.GraphReaderTest`). Details are elided.

JDOM JDOM is a Java-based document object model library for XML. The JDOM test suite (version 1.x) includes 16 test classes in 3 packages. Its tests often use try/catch blocks to ensure that exceptions are thrown as expected. Assertion fingerprints' inclusion of try/catch blocks thus work particularly well for JDOM—its true positive rate is (85% ± 9%). Figure 3.3 shows an example where a `fail(String)` at the end of a try block ensures that some prior statements throws the expected exception.

```
1  public void test_TCC___String() {
2    // [... 4x assertTrue(String, boolean)]
3    try {
4      // ...
5      fail("allowed creation of an element with no name");
6    } catch (IllegalNameException e) { }
7    try {
8      // ...
9      fail("allowed creation of an element with null name");
10   } catch (IllegalNameException e) { }
11 }
```

```
1  public void test_TCC___String_String() {
2    // [... 4x assertTrue(String, boolean)]
3    try {
4      // ...
5      fail("allowed creation of an element with no name");
6    } catch (IllegalNameException e) { }
7    try {
8      // ...
9      fail("allowed creation of an element with null name");
10   } catch (IllegalNameException e) { }
11 }
```

Figure 3.3: JDOM uses try/catch/`fail()` to ensure certain exceptions are thrown.

Joda-Time Joda-Time is a library for manipulating dates and times. The Joda-Time test suite (version 2.0) consists of 121 test classes in 6 packages. The samples showed that the Joda-Time test suite contains many test method clones with straight-line assertion

fingerprints, as seen for instance in Figure 3.4. The depicted clone set includes tests `Test{Buddhist,Coptic,Ethiopic,GJ,Gregorian}Chronology`, all of which have identical structure and can be refactored using theories. This example also shows that the *lack* of control-flow structures can also be a characteristic used by assertion fingerprints to match test methods.

```
1  public void testEquality() {
2    assertSame(BuddhistChronology.getInstance(TOKYO),
3               BuddhistChronology.getInstance(TOKYO));
4    assertSame(BuddhistChronology.getInstance(LONDON),
5               BuddhistChronology.getInstance(LONDON));
6    assertSame(BuddhistChronology.getInstance(PARIS),
7               BuddhistChronology.getInstance(PARIS));
8    assertSame(BuddhistChronology.getInstanceUTC(),
9               BuddhistChronology.getInstanceUTC());
10   assertSame(BuddhistChronology.getInstance(),
11              BuddhistChronology.getInstance(LONDON));
12 }
```

(a) `testEquality()` for `TestBuddhistChronology`.

```
1  public void testEquality() {
2    assertSame(CopticChronology.getInstance(TOKYO),
3               CopticChronology.getInstance(TOKYO));
4    assertSame(CopticChronology.getInstance(LONDON),
5               CopticChronology.getInstance(LONDON));
6    assertSame(CopticChronology.getInstance(PARIS),
7               CopticChronology.getInstance(PARIS));
8    assertSame(CopticChronology.getInstanceUTC(),
9               CopticChronology.getInstanceUTC());
10   assertSame(CopticChronology.getInstance(),
11              CopticChronology.getInstance(LONDON));
12 }
```

(b) `testEquality()` of `TestCopticChronology`.

Figure 3.4: Two implementations of `testEquality()` make up one clone set from the Joda-Time test suite. Both contain straight-line assertion fingerprints.

Apache Commons Collections The Apache Commons Collections augments the Java Collections Framework with additional data structures. The 151 test classes in the 11 packages of the Apache Commons Collections test suite (version 3.3) include a number of refactorable clones as tests. Furthermore, some test methods are exactly identical (modulo whitespace) but reside in different classes and hence invoke different helper code. In Figure 3.5, test methods `testIterator()` of classes `Test{ArrayIterator, IteratorChain, ObjectArrayIterator, UniqueFilterIterator}` are textually identical. However, their containing classes implement different `makeFullIterator()` methods and `testArray` arrays. Hence, the tests exercise different data structures despite being identical. This implementation pattern makes the test methods in the Apache Commons Collection highly refactorable.

```
1  public void testIterator() {
2    Iterator iter = (Iterator) makeFullIterator();
3    for (int i = 0; i < testArray.length; i++) {
4      Object testValue = testArray[i];
5      Object iterValue = iter.next();
6      assertEquals("Iteration value is correct", testValue, iterValue);
7    }
8    assertTrue("[...] should now be empty", !iter.hasNext());
9    try {
10     Object testValue = iter.next();
11   } catch (Exception e) {
12     assertTrue(
13        "NoSuchElementException must be thrown",
14        e.getClass().equals((new NoSuchElementException()).getClass()));
15  }
16 }
```

Figure 3.5: Test method from the Apache Commons Collections, belonging to classes `Test{ArrayIterator, IteratorChain, ObjectArrayIterator, UniqueFilterIterator}`. At runtime, `makeFullIterator()` (line 2) and the value of `testArray` (line 3, 4) bind different values depending on the containing class.

Weka Weka is a machine learning software suite. The Weka test suite (version 3.7.8) consists of 98 test classes in 17 packages. Weka has classes that im-

plement similar machine learning techniques in both supervised and unsupervised variants. One example is `weka.filters.supervised.attribute.DiscretizeTest` vs `weka.filters.unsupervised.attribute.DiscretizeTest`. Both tests exercise the methods of class `Discretize` but one targets the supervised version and the other targets the unsupervised version. Test method `testTypical()` is textually identical between the supervised and unsupervised variants of `DiscretizeTest`. One can refactor this clone set by theorizing data points of supervised and unsupervised `Discretize` objects and applying the same test method.

JFreeChart JFreeChart is a library allowing the creation and display of professional quality charts in applications. The JFreeChart test suite (version 1.0.15) consists of 348 test classes in 23 packages, and in particular contains test method clones which test related types. Figure 3.6 shows that the test method named `testClear()` in classes `VectorSeriesTests` and `XIntervalSeriesTests` are identical but perform tests on `VectorSeries` and `XIntervalSeries`, respectively. (Unlike with the Commons Collection, `testClear()` only differs in terms of the calls to the class under test, not to self-calls on the test class.) Theories would be straightforward to apply to such test methods. Despite testing different data types, the example in Figure 3.6 appears to be refactorable.

Apache POI Apache POI is a Java API for Microsoft documents. Our technique worked poorly on this benchmark, yielding a low true positive rate of $(30 \pm 20)\%$ on the 50 test classes in 10 packages of Apache POI test suite (version 3.9). Figure 3.7 shows an example where, despite having the same ordered set of assertions and control-flow fingerprints, methods `testBasics()` and `testImageCount()` are not clones and are completely irrelevant. Hence, the Apache POI test suite exposes the limitations (Subsection 4.3) of assertion fingerprints. Specifically, because our technique does not gather information on assertion parameters, it cannot detect the differences between the arguments passed to different calls to `assertNotNull` and `assertEquals(int, int)`in Figure 3.7, making it susceptible to a certain class of false positives.

Google Visualization Data Source Library The Google Visualization Data Source Library enables the visualization of data sources. The test suite (version 1.1.2) contains 50 test classes in 10 packages. We found that this suite contains test methods involving complex query-related statements, consequently reducing the roles of assertions in a test method. Furthermore, this test suite also uses custom helper methods such as `void assertStringArraysEqual(String[] expected, String[] found)` in

```
1  public void testClear() {
2    ComparableObjectSeries s1;
3    // (a) in VectorSeriesTests:
4    // s1 = new VectorSeries("S1");
5    // (b) in XIntervalSeriesTests:
6    // s1 = new XIntervalSeries("S1");
7    s1.addChangeListener(this);
8    s1.clear();
9    assertNull(this.lastEvent);
10   assertTrue(s1.isEmpty());
11   s1.add(1.0, 2.0, 3.0, 4.0);
12   assertFalse(s1.isEmpty());
13   s1.clear();
14   assertNotNull(this.lastEvent);
15   assertTrue(s1.isEmpty());
16 }
```

Figure 3.6: JFreeChart contains `testClear()` methods which are identical except for the type under test.

`com.google.visualization.datasource.query.engine.QueryEngineTest`. Google Visualization's test suite thus exploits limitations of our technique, resulting in a below-average true positive rate of $(60 \pm 15)\%$.

3.3 Large Clone Sets

Figure 3.1 shows that 5 of our clone sets include over 100 methods each. These large sets have between 1 to 9 assertions, and belong to the Joda-Time (3 sets) and JFreeChart (2 sets) test suites. We investigated them in more detail, and found that a majority of the reported clones in these large clone sets were indeed true positive clones; 1 of the reported clone sets was fully true positive, while another 3 sets were fragmented true positives, with a majority of the contents of those sets being true positives.

The JFreeChart true positive clone set finds 151 methods with 3 matching `assertTrue(boolean)` calls; these calls succeed 1, 2, and 2 control-flow merges respectively (Figure 3.8). Manual inspection revealed that JFreeChart uses exactly the same

```
1  public void testBasics() throws Exception {
2    EscherStm es;
3    EscherDelayStm eds;
4    // ...
5    assertNotNull(es);
6    assertNotNull(eds);
7    assertEquals(13, es.getEscherRecords().length);
8    assertEquals(0, eds.getEscherRecords().length);
9  }
```

(a) `testBasics()` for `org.apache.poi.hpbf.model.TestEscherParts`.

```
1  public void testImageCount() {
2    HWPFDocument docA, docB;
3    // ...
4    assertNotNull(docA.getPicturesTable());
5    assertNotNull(docB.getPicturesTable());
6    List<Picture> picturesA, picturesB;
7    // ...
8    assertEquals(7, picturesA.size());
9    assertEquals(2, picturesB.size());
10 }
```

(b) `testClear()` for `org.apache.poi.hwpf.TestHWPFPictures`.

Figure 3.7: A false positive clone set from the Apache POI's test suite. Not considering the arguments passed to the assertions causes this false positive.

code to implement a method called `testCloning()` (148) or `testCloning2()` (3) across different classes.

The remaining 3 clone sets include one from JFreeChart and 2 from Joda-Time. The JFreeChart set contains 124 methods with 2 straight-line asserts (`assertTrue(boolean)` succeeded by `assertEquals(int, int)`), 99% (123/124) of which are all implementing `testHashcode()` with the same code. The Joda-Time examples are testing arithmetic operations and constructors with identical code; a few false positives sneak into these sets.

The one clone set containing false positives is a 201-method clone set with one assert inside a try. The assert is simply a call to `fail()`, with no message. This could be

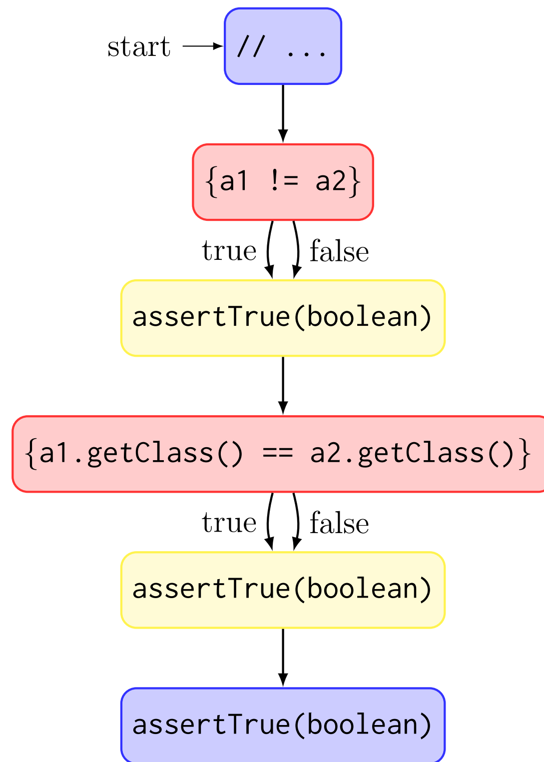

Figure 3.8: A control-flow graph for method `testCloning()` in the JFreeChart test suite. Irrelevant details elided: we only include assertions and predicates.

considered bad practice—there is no error message indicating the reason for the failure. This clone set is the only result that is very fragmented and contains smaller clone sets.

Chapter 4

Discussion

We next discuss several aspects of our results in greater depth. Previously, we presented counts of false and fragmented true positives; in this section, we discuss some potential causes of these non-true positive results. We also qualitatively discuss the refactorability of our reported clone sets. We continue by comparing our results with clone detectors and conclude this section with a survey of limitations and threats to validity.

4.1 False and Fragmented True Positives

While our technique returns many methods that are indeed clones, it also returns some false positives (Definition 8) and some fragmented true positives (Definition 9).

False Positives False positives occur when two different methods have, by chance, the same assertion fingerprints. For instance, POI contains two methods, `testBasics()` and `testImageCount()`, which both have 4 straight-line assertions of the same types, but no conceptual similarity.Figure 3.7 is an example of a false positive. False positive clone sets are undesirable. Fortunately, they appear relatively rarely: they account for $(10 \pm 5)\%$ of the manually-investigated random sample and 13% of the filtered clone sets.

Fragmented True Positives Recall that fragmented true positives occur when at least one method in a clone set is not a clone of some other method in the same clone set. For instance, one clone set in Commons Collections includes four methods with the same

assertion fingerprint—`fail()` with 1 exceptional successor. (See Figure 4.1.) This clone set confounds two 2-element clone sets. (A more complex clone detection algorithm could split such clone sets; however, simplicity was one of our design goals.) While fragmented true positives help understand the underlying prevalence of cloning in a benchmark suite, their lack of unity makes them difficult to work with and they are hence likely to be lower on the priority list for potential refactoring. We therefore believe that it is helpful to the developer to filter out fragmented true positives whenever possible, and indeed, the clone sets removed by our filtering (Table 3.4) are 63% fragmented true positive.

4.2 Refactorability of Clones

Because our technique uses assertion fingerprints to group similar test methods, methods in the same clone set will certainly be similar in structure. (See Figures 2.1, 3.2, 3.3, 3.4, 3.5, 3.6 for examples.) This similarity should be helpful for refactoring. In this work, we are agnostic as to refactoring technology, but possibilities include helper methods, parametrized unit tests, and theories. Nevertheless, some clone sets remain difficult to refactor. This difficulty surely contributes to the existence of clones in benchmark suites.

Non-parametrizable Clones Figure 4.2 demonstrates a non-parametrizable clone set. Despite similar structures, `testValueList_getByIndex()` tests the `get(int)` method of a `ListOrderedMap` object by iterating forward, whereas `testValueList_removeByIndex` removes items from a `ListOrderedMap` object while iterating until the `ListOrderedMap` object only contains one element. The difference in semantics makes the two test methods difficult or impossible to refactor.

Ubiquitous Clones Ubiquitous clones are short clones with high frequency across a system [9]. We have discovered a few ubiquitous clones in our results. For example, copies of the 4-line test method `testToolTips` (Figure 4.3) exist in 10 test classes of the Weka test suite. Ubiquitous clones appear to be methods that perform short, specific tasks, or that were already refactored with helper methods and are now short. Ubiquitous clones are not only difficult to refactor because of their highly frequent appearance in the system, but also unlikely worth the effort to refactor because they are so short.

4.3 Limitations

The key assumption behind our technique is that test methods with the same assertion fingerprints are likely to be clones. Test suites that do not respect this assumption expose the limitations of our clone detection technique. Specifically, our technique works less well when the specific arguments to the assertion are important, or when assertions have already been refactored out into helper methods. More generally, assertion fingerprints work best when the ordered set of assertions in a method is important, and less well when tests use other means to verify program behaviour.

Assertion Arguments Assertion fingerprints focus on the structure of the test methods and not on the data flowing to the particular assertions. We observed that this worked well for most of our test suites. However, Apache POI's test suite (Figure 3.7) would require finer-grained information for best results.

Nevertheless, certain refactorable clones have similar structure but test different data types; recall, for instance, JFreeChart (Figure 3.6). Consequently, we designed our clone detection technique to choose to detect more true positive clones by reporting clones that involve differently-typed values as assertion arguments. However, this also causes the technique to report some false positives with differently-typed assertion arguments. (In other words, matching assertions with different data types increases our recall at the cost of lower precision.)

Helper Methods Occasionally, test methods are already somewhat refactored and use helper methods to invoke relevant assertions. Our technique currently relies on bare assertions appearing in test methods. Consider, for instance, Figure 4.4. It shows that `void testBuildInputSource()` uses helper method `assertXMLMatches`. Our technique is unaware of helper methods' effects and does not include the assertions of callees in assertion fingerprints, which may cause additional false positives. Test methods that already use helper methods are unlikely to be worth the effort to refactor.

4.4 Comparison with Clone Detectors

We compare our technique to traditional clone detectors, both in principle, and empirically, via a head-to-head comparison with the best-in-class clone detector Bauhaus [5].

Contextual Information As a byproduct of computing assertion fingerprints, our technique has additional information about why certain methods belong to clone sets. We make this information available to the developer. We believe that it will enable developers to better refactor the reported clones.

In particular, our tool reports the predicates contributing to each assertion's branch count and merge count. It also reports quantitative information—counts of methods, classes, and packages for each clone set—that could hint at the relevance and refactorability of the reported clones.

Comparison with Bauhaus To further evaluate our results, we performed a head-to-head comparison of our clone sets with those from Bauhaus. Unlike our technique, Bauhaus operates by comparing tokens at source level and detects similar patterns throughout a codebase. We ran Bauhaus with a token threshold of 50 (normal for Bauhaus). Bauhaus's performance is subject to tuning the token threshold; a lower value detects more clones and also gives more spurious results. Our technique does not require thresholds. Recall that our technique works at method level and returns function clones, while Bauhaus is oblivious to code structure and detects duplicate code segments within methods.

In particular, we empirically assessed how well Bauhaus performed against Assertion Fingerprints (how many clones from our results were found and how many were missed by Bauhaus, Table 4.1) and vice versa (Table 4.2). We inspected randomly selected results from Bauhaus and found that:

- because Bauhaus is insensitive to method boundaries, it detects more clone sets (5367) than we do (978). However, Bauhaus clone sets therefore include arbitrary method fragments. We believe that our function clones are easier to refactor and more relevant to developers; returning function clones also helps avoid spurious results, particularly ubiquitous clones.

- our technique is unaffected by mid-clone textual differences that Bauhaus is sensitive to, as we operate on Java bytecode rather than source code. (For instance, Bauhaus would not detect a clone with 25 identical tokens, a different token, and then 25 more identical tokens.) Hence, some of our clone sets are supersets of Bauhaus clone sets.

- our technique detected 443 clone sets (45% of our results) that Bauhaus missed. In particular, Bauhaus (with threshold 50) was not able to detect our motivating example (Figure 2.1).

Table 4.1: How Bauhaus performed against Assertion Fingerprints—45% of clone sets from Assertion Fingerprints were not detected by Bauhaus.

	Clone sets also detected by Bauhaus				Clone sets not
	Type 1	Type 2	Type 3	Total	detected by Bauhaus
Apache POI	40	6	8	54	131
Commons Collections	88	5	18	111	55
Google Visualization	11	1	0	12	23
HSQLDB	19	11	4	34	31
JDOM	0	0	0	0	29
JFreeChart	49	9	36	94	52
JGraphT	2	2	2	6	5
JMeter	31	5	9	45	15
Joda-Time	85	18	39	142	89
Weka	28	5	4	37	13
Total	353	62	120	535	443

- our technique does miss clones where assertions get added or removed, as it requires exact matches on assertion fingerprints. Bauhaus can find such clones by matching around the missing assertions. Hence, some of our clone sets are subsets of Bauhaus clone sets. (We tried matching assertion fingerprint sets with edit distance of 1, but that caused too many false positives.) Requiring exact matches increases the precision of our technique at the cost of decreased recall.

Furthermore, 45% of our clone sets were not detected by Bauhaus. Our technique captures additional, high-quality, clones which are not reported by existing clone detectors.

4.5 Threats to Validity

The selection of 10 benchmarks from different application areas (graph theory, machine learning, data structures, etc) aims to mitigate threats to external validity. However, no benchmark suite is exhaustive and captures all possible test code styles, especially those specific to certain applications or domains. Also, our benchmark suite size does not

Table 4.2: How Assertion Fingerprints performed against Bauhaus—Assertion Fingerprints found 49% of clone sets detected by Bauhaus.

	Clone sets also detected by A.F.				Clone sets not detected by A.F.			
	Type 1	Type 2	Type 3	Total	Type 1	Type 2	Type 3	Total
Apache POI	156	12	18	186	330	20	24	374
Commons Collections	274	15	32	321	174	7	15	196
Google Visualization	29	1	0	30	81	4	16	101
HSQLDB	55	6	2	63	815	19	52	886
JDOM	0	0	0	0	112	6	8	126
JFreeChart	432	39	33	504	409	21	43	473
JGraphT	8	2	4	14	18	0	0	18
JMeter	129	9	8	146	117	3	11	131
Joda-Time	990	123	120	1233	343	11	26	380
Weka	123	10	5	138	44	3	0	47
Total	2196	217	222	2635	2443	94	195	2732

exceed 270 kLOC (lines of code in the test suites). Larger benchmarks may have different properties than those in our set.

We have not investigated non-Java or non-JUnit benchmarks. Our results might not generalize to such test suites.

The major threat to internal validity in our case is from confounding effects. Our results show that similar assertion fingerprints are correlated with test method clones. However, the assertion fingerprints may simply be reflecting some other property of the code.

A threat to construct validity is that the clone sets were manually analyzed, using a subjective 3-point scale (true positive, fragmented true positive, false positive).

We believe that we have adequately mitigated the threats to validity through benchmark selection and the use of definitions of clone characteristics. The result is an accurate assessment of our technique's performance on JUnit suites.

4.6 Future Work

We would like to investigate more test suite benchmarks and corresponding results to devise our technique and further improve true positive rates and coverage.

Based on the clones detected with our technique, we would like to explore opportunities in automatic refactoring (e.g. parametrizing) refactorable clone sets to reduce developers' effort, perhaps with existing techniques ([4, 2, 13]).

We would also like to explore beyond assertions but static method invocations in general. In fact, assertions are static methods of class `junit.framework.Assert`. We would like to examine static methods of, for example, libraries such as math and algorithms (or application programming interfaces, APIs) and investigate how effective the static methods can be used to detect clones in a software system. Our assumption of detecting test method clones is that assertions play major roles in test methods. Lacking knowledge of to what extend each static method is involved in a software system, we could use techniques such as machine learning (e.g. [1]) to approximate the importance of static methods around which we detect software clones in a similar fashion (e.g. matching control flow fingerprints).

```
1 public void testAddToFullBufferNoTimeout() {
2   final Buffer bounded = BoundedBuffer.decorate(new UnboundedFifoBuffer(),
        1);
3   bounded.add( "Hello" );
4   try {
5     bounded.add("World");
6     fail();
7   } catch (BufferOverflowException e) { }
8 }
```

(a) `testAddToFullBufferNoTimeout` for `org.apache.commons.collections.buffer.TestBoundedBuffer`

```
1 public void testAddAllToFullBufferNoTimeout() {
2   final Buffer bounded = BoundedBuffer.decorate(new UnboundedFifoBuffer(),
        1);
3   bounded.add( "Hello" );
4   try {
5     bounded.addAll(Collections.singleton("World"));
6     fail();
7   } catch (BufferOverflowException e) { }
8 }
```

(b) `testAddAllToFullBufferNoTimeout` for `org.apache.commons.collections.buffer.TestBoundedBuffer`

```
1 public void testConstructorEx() throws Exception {
2   try {
3     new LoopingIterator(null);
4     fail();
5   } catch (NullPointerException ex) { }
6 }
```

(c) `testConstructorEx` for
`org.apache.commons.collections.iterators.TestLoopingIterator`

```
1 public void testConstructorEx() throws Exception {
2   try {
3     new LoopingListIterator(null);
4     fail();
5   } catch (NullPointerException ex) { }
6 }
```

(d) `testConstructorEx` for
`org.apache.commons.collections.iterators.TestLoopingListIterator`

Figure 4.1: Four test methods of a fragmented true positive clone set from the Apache Commons Collections test suite. Despite all having the same assertion fingerprint, only the first two methods are clones, and the last two methods are clones.

```
1 public void testValueList_getByIndex() {
2   resetFull();
3   ListOrderedMap lom = (ListOrderedMap) map;
4   for (int i = 0; i < lom.size(); i++) {
5     Object expected = lom.getValue(i);
6     assertEquals(expected, lom.valueList().get(i));
7   }
8 }
```

```
1 public void testValueList_removeByIndex() {
2   resetFull();
3   ListOrderedMap lom = (ListOrderedMap) map;
4   while (lom.size() > 1) {
5     Object expected = lom.getValue(1);
6     assertEquals(expected, lom.valueList().remove(1));
7   }
8 }
```

Figure 4.2: A non-parametrizable clone set from the Apache Commons Collections test suite.

```
1 public void testToolTips() {
2   if (!m_GOETester.checkToolTips())
3     fail("Tool tips inconsistent");
4 }
```

Figure 4.3: A member of an ubiquitous clone set from Weka. Identical copies of `testToolTips()` exist in 10 test classes.

```
1  private void assertXMLMatches(String baseuri, Document doc) {
2    // Performs a series of assertions ...
3  }
4
5  public void testBuildInputSource() {
6    InputSource is;
7    // ...
8    try {
9      assertXMLMatches(null,
10       new SAXBuilder().build(is));
11   } catch (Exception e) {
12      // ...
13   }
14 }
```

Figure 4.4: A JDOM test method using a helper method. Helper method assertions are invisible to our technique.

Chapter 5

Related Work

In prior work, Jain [6] developed Assertion Protocols, a technique that collects histories such as definitions and method calls made by the assertion arguments. Based on the histories of assertion arguments, Jain used a scoring algorithm to determine similarity levels between assertions. Unfortunately, Jain's technique suffered from a high false positive rate on the same benchmark suite in this thesis. As a result, we employed a different approach that focuses on assertions fingerprints (test method structures) and is oblivious to assertion arguments.

Demme and Sethumadhavan developed a method to characterize programs by using agglomerative clustering on low level representations, including dynamic data flow graphs and static control flow graphs [3]. They used this technique to identify unique code behaviours for program optimization. Demme and Sethumadhavan's technique computes graphs and approximates comparisons, whereas ours approximates graphs and performs precise comparisons. Nonetheless, their technique also discovers that "static control/data flow graphs are very useful for identifying identical functions", which is congruent with our discoveries.

More broadly, assertion fingerprints were inspired by the development of parametrized unit tests and theories, as described in Section 1. Fraser and Zeller presented an approach to generate parametrized unit tests with test generation and mutation given a concrete test method [4]. Zhang et al proposed a combined dynamic and static automated test generation approach to improve structural coverage [15].

Static analysis has been used to discover constraints and common design patterns. Whaley et al proposed an approach that includes static analysis to model finite state machines in a software system [14]. Ammons et al used both static analysis and machine learning to

discover program behaviours and by observing interactions with application programming interfaces (APIs) or abstract datatypes (ADTs) [1]. The output of [1] is a specification automaton that our technique may feed on because we also focus on interactions with APIs (assertions) but in the context of test suites.

Recent research has also investigated automated and guided refactoring. However, our focus on testing code is novel. Balazinska et al proposed an approach to refactor common parts of method clones, parametrizing their differences [2]. Their technique is limited to certain types of applicable clones. An alternative approach proposed by Volanschi [13] also covers other domain-specific and application-specific clones but requires manual intervention. Because we focus on test code, our technique would require less manual effort than Volanschi's technique on our domain.

Chapter 6

Conclusions

This thesis presents a technique for detecting clones in test suites. Our technique collects assertion fingerprints, which summarize the control-flow surrounding assertion and fail calls in test methods. It then partitions test methods into clone sets according to ordered sets of assertion fingerprints.

Assertion fingerprints contain 5 control-flow components: a branch count; a merge count; an exceptional successors count; and two boolean flags, indicating whether the given assertion is in a loop or a catch block. Matching test methods based on these 5 components allows the matched methods to possess structural similarities in assertion calls and therefore be amenable to refactoring.

We implemented our technique and collected clone sets for 10 open-source Java test suites, analyzing them via empirical study and qualitative analysis. Empirically, 44% of test methods in our suite were reported as clones; and clone sets involve a median of 3 methods, 2 packages, and 1 class. Qualitatively, some test suites do indeed have textually identical clones of methods with similar data types (e.g. Weka, Apache Commons Collections); exception flow is important for clone detection; and that reported cloned methods operating on different data types may either be true positives or false positives. Our results show that our technique works well to find clone sets which are amenable to refactoring, with a tolerable false positive rate.

APPENDICES

Appendix A

Output Sample

```
[<setID>] Set (packages: 1, classes: 1):
Assertions (1):
  <junit.framework.TestCase: void assertTrue(boolean)> (bc:1, mc:1, es:0,
     inLoop:true, inCatch:false)

Methods (4):
  <weka.filters.unsupervised.attribute.RemoveTypeTest: void
     testNominalFiltering()>
    <weka.filters.unsupervised.attribute.RemoveTypeTest.java:65> (predicates
       :{(i0 < $i2)=true, ($i1 == 1)=null})
  <weka.filters.unsupervised.attribute.RemoveTypeTest: void
     testStringFiltering()>
    <weka.filters.unsupervised.attribute.RemoveTypeTest.java:73> (predicates
       :{(i0 < $i2)=true, ($i1 == 2)=null})
  <weka.filters.unsupervised.attribute.RemoveTypeTest: void
     testNumericFiltering()>
    <weka.filters.unsupervised.attribute.RemoveTypeTest.java:81> (predicates
       :{(i0 < $i2)=true, ($i1 == 0)=null})
  <weka.filters.unsupervised.attribute.RemoveTypeTest: void
     testDateFiltering()>
    <weka.filters.unsupervised.attribute.RemoveTypeTest.java:89> (predicates
       :{(i0 < $i2)=true, ($i1 == 3)=null})
```

Figure A.1: Output display of clone set in Figure 2.1.

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 4–16, 2002.

[2] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 326–336, 1999.

[3] John Demme and Simha Sethumadhavan. Approximate graph clustering for program characterization. In *ACM Transactions on Architecture and Code Optimization (TACO) - HIPEAC*, volume 8. ACM New York, NY, USA, 2012.

[4] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374, 2011.

[5] Nils Gode. Evolution of type-1 clones. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, 2009.

[6] Divam Jain. Detecting test clones with static analysis. Master's thesis, University of Waterloo, 2013.

[7] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, 2009.

[8] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[9] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, Kingston Ontario, Canada, September 2007.

[10] David Saff. Theory-infected: Or how I learned to stop worrying and love universal quantification. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 846–847, 2007.

[11] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 294–309, 2011.

[12] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 253–262, 2005.

[13] N. Volanschi. Safe clone-based refactoring through stereotype identification and iso-generation. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 50–56, 2012.

[14] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, 2002.

[15] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363, 2011.