Detecting Test Clones with Static Analysis

by

Divam Jain

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

Waterloo, Ontario, Canada, 2013

Abstract

Large-scale software systems often have correspondingly complicated test suites, which are difficult for developers to construct and maintain. As systems evolve, engineers must update their test suite along with changes in the source code. Tests created by duplicating and modifying previously existing tests (clones) can complicate this task.

Several testing technologies have been proposed to mitigate cloning in tests, including parametrized unit tests and test theories. However, detecting opportunities to improve existing test suites is labour intensive.

This thesis presents a novel technique for detecting similar tests based on type hierarchies and method calls in test code. Using this technique, we can track variable history and detect test clones based on test assertion similarity.

The thesis further includes results from our empirical study of 10 benchmark systems using this technique which suggest that test clone detection by our technique will aid test de-duplication efforts in industrial systems.

Acknowledgements

I would like to thank Patrick Lam and Reid Holmes for their guidance and advice, without which this work would not be possible. I would also like to thank Xavier Noumbissi for his invaluable help navigating the intricacies of the Soot framework and Wei Wang for his discussions on clone detection.

Dedication

My grandparents introduced me to the world of computers and fostered my fascination with how they work. For this, I dedicate this thesis to them.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

2 Background and Definitions **6**

2.1 Unit Testing in Java 6

2.2 Soot and Static Analysis 8

2.3 Summary 9

3 Clone Detection Approach **11**

3.1 Assertion Protocol Generation 14

3.2 Similarity Analysis 16

3.3 Result Consolidation 17

3.4 Final Display 19

3.5 Conclusion of running example 20

3.6 Summary 20

4 Results of Empirical Study **23**

4.1 Experimental Results 24

4.2 General Trends 25

4.3 Developer impressions 26
4.3.1 JodaTime 26
4.3.2 Apache Commons-Collections 27
4.3.3 JFreeChart 28
4.3.4 HSQLDB 29
4.3.5 JGraphT 29
4.3.6 jDom 29
4.3.7 Weka 30
4.4 Summary 30

5 Discussion **31**
5.1 Scope of Our Analysis 31
5.2 Retrofitting vs. Early Optimization 32
5.3 Comparison with Clone Detectors 32
5.4 Result Quality 33
5.5 Threats to Validity 33
5.6 Limitations 34
5.7 Future work 35
5.8 Summary 36

6 Related Work **37**
6.1 Alternative Testing Techniques 38
6.2 Code Clone Detection 38
6.3 Refactoring 41
6.4 Program Slicing 41

7 Conclusions **43**

APPENDICES **44**

A Data Flow Analysis Definition 45

B Detailed Test Suite Information 47

C Output Sample 48

References 50

List of Tables

4.1 Benchmark systems. 23

4.2 Test clones reported by our analysis. 24

B.1 Code size, test case count (as reported by JUnit) and analyzed test method count. 47

List of Figures

1.1 2 of 9 test clones as found in Jodatime. Each comes from a different unit's tests and is identical in every way except the unit name. 4

1.2 JodaTime tests from Figure 1.1 written as a unified theory. 5

2.1 Sample unit test for illustrating JUnit testing pattern. 7

2.2 Jimple 3-address code version of the setUp method. 9

2.3 Jimple 3-address code version of the test method. 10

2.4 Jimple 3-address code version of the tearDown method. 10

3.1 Simple system under test: Class hierarchy with abstract collection and two implementations. 12

3.2 Concrete Unit Tests (CUTs) for Collection class subtypes. 12

3.3 Rules used in Assertion Protocol Generation (at assignment statements). . 14

3.4 An example of branch handling during AP Generation. 15

3.5 Assertion Protocols Generated for list test in running example (Figure 3.2. 21

3.6 Graphical view of assertions (nodes) and their similarity (edges, which denote assertions whose similarity exceeds the threshold). 21

3.7 Final results displayed for the running example by the implementation. . . 22

3.8 Expected theory based tests for the Collection class' children. 22

4.1 Shared behaviour testing in Commons Collections. 27

4.2 Two similar testAddSeries() implementations in jFreeChart. 28

4.3 An implicit unit test in HSQLDB. 29

A.1 This is the formal definition of the data flow analysis as implemented in Soot. 46

C.1 Output sample from our tool running on JodaTime. 49

Chapter 1

Introduction

Unit tests are a popular technique for automated testing used to improve and maintain code quality [32]. Ideally, tests in a suite work together to cover all the functionality of the single compilation unit. Unit tests are traditionally designed to be self contained and run independently, taking no input parameters. An example of a typical unit test is given in Chapter 2.

In well-known unit testing frameworks such as JUnit, achieving acceptable coverage requires writing repetitive boilerplate code to make the tests self-contained which is only mitigated at a per unit level [6]. This boilerplate code leads to code duplication, increasing maintenance overheads and possibly propagating errors due to inconsistent modification in individual tests [18].

Since the setup and teardown methods which allow for code reuse in tests (discussed in Section 2.1) only operate at the unit level, a developer may resort to cloning test code from a sibling class and extending it. Figure 1.1 shows 2 of 9 test cases in the JodaTime library which are identical in every way other than the module under test. In general, such manually copied code increases maintenance overheads and is considered undesirable.

We observed such clones in our empirical study, described with greater detail in Chapter 4 A variety of techniques have been proposed to alleviate this problem. Tillman and Schulte have proposed *parametrized unit tests (PUTs)* to increase the expressive power of unit testing and to enable test reuse [38]. Along similar lines, Saff proposed *Theories* to simplify and increase the robustness of unit tests [34]. Other techniques to reduce code duplication leverage language features such as inheritance, generics and decomposition, to reuse test code. A developer can use these techniques to remove the clones like those from Figure 1.1.

Developers using Theories can refactor tests and eliminate the clones as seen in Figure 1.2. Here, the units to be tested are a parameter to the test and the @DataPoints annotation provides the list of suitable arguments. As a result of using a Theory the code size is reduced from 126 lines of 9 Type-2 clones to 27 lines of non-cloned code.

Retrofitting existing test suites to reduce undesirable clones is clearly valuable. According to Saff, the use of theories reduces the long term maintenance cost for test suites [33]. Moreover, Thummalapenta et al. conclude from an empirical study of existing test suites that parametrization is beneficial–their results indicate that test suites retrofitted with parametrization can detect new defects and provide increased branch coverage [37]. Any of the techniques mentioned above would also make the tests less brittle and easier to understand.

While writing new tests using these techniques has been shown to be beneficial, it is not always clear when and where to apply them. During initial development, a simple unit test can be adequate. However, this design may become incrementally more difficult to maintain as the code base evolves leading to the situation described above.

The problem with retrofitting existing test code using these techniques is that first, identifying situations where refactoring would be appropriate is hard. Most clone detection tools operate on symbols in source code. Hence, even while detected clones may be suitable for retrofitting, the results do not provide a signal to help decide if or how the clone should be refactored.

Second, the feasibility of implementing the refactoring manually is limited by the scarcity of developer time [9, 30]. In particular, identifying the solution to avoid clones is a time consuming process and is ill-suited for developers who are likely to have higher priority tasks than addressing technical debt in test code.

Our goal is to reduce the developer effort needed for these refactorings. We present a technique to automatically detect test clones and provide contextual information that may assist in the refactoring process. The technique we used to detect clones is described in Chapter 3.

We implemented our technique using the Soot program analysis framework [24]. The tool we created analyzes code and displays the set of test methods that verify one or more similar properties of the code along with specific program properties used to draw this inference. Details about the output are described in Chapter 3.4.

Refactoring the appropriate test methods will result in a test suite that contains fewer clones and is easier to modify when developers change code in the system under test.

We have implemented our technique using the Soot program analysis framework [24]. Our evaluation of the implementation is based on a suite of 10 benchmark programs ranging from 8,000 to 246,000 lines of code. Our technique identified from 20.21% to 54.61% clones in members of our benchmark suite. We manually inspected 10 randomly sampled clones from these tests and found that our recommendations were suitable clones and often amenable to refactoring. Detailed findings are in Chapter 4.

Our primary contributions are:

- a characterization of a class of test-level clones that are suitable for refactoring;
- a technique for identifying such test-level clones in existing test suites;
- an implementation of that technique; and
- an empirical study of a significant suite of benchmark programs using our technique.

```
public void testDurationFields() {
assertEquals("eras", GJChronology.getInstance().eras().getName());
assertEquals("centuries", GJChronology.getInstance().centuries().getName());
assertEquals("years", GJChronology.getInstance().years().getName());
assertEquals("weekyears", GJChronology.getInstance().weekyears().getName());
assertEquals("months", GJChronology.getInstance().months().getName());
assertEquals("weeks", GJChronology.getInstance().weeks().getName());
assertEquals("halfdays", GJChronology.getInstance().halfdays().getName());
assertEquals("days", GJChronology.getInstance().days().getName());
assertEquals("hours", GJChronology.getInstance().hours().getName());
assertEquals("minutes", GJChronology.getInstance().minutes().getName());
assertEquals("seconds", GJChronology.getInstance().seconds().getName());
assertEquals("millis", GJChronology.getInstance().millis().getName());
}
```

(a) testDurationFields() for GJChronology

```
public void testDurationFields() {
assertEquals("eras", ISOChronology.getInstance().eras().getName());
assertEquals("centuries", ISOChronology.getInstance().centuries().getName());
assertEquals("years", ISOChronology.getInstance().years().getName());
assertEquals("weekyears", ISOChronology.getInstance().weekyears().getName());
assertEquals("months", ISOChronology.getInstance().months().getName());
assertEquals("weeks", ISOChronology.getInstance().weeks().getName());
assertEquals("days", ISOChronology.getInstance().days().getName());
assertEquals("halfdays", ISOChronology.getInstance().halfdays().getName());
assertEquals("hours", ISOChronology.getInstance().hours().getName());
assertEquals("minutes", ISOChronology.getInstance().minutes().getName());
assertEquals("seconds", ISOChronology.getInstance().seconds().getName());
assertEquals("millis", ISOChronology.getInstance().millis().getName());
}
```

(b) testDurationFields() for ISOChronology

Figure 1.1: 2 of 9 test clones as found in Jodatime. Each comes from a different unit's tests and is identical in every way except the unit name.

```
  @Theory
  public void testDurationFields(BaseChronology chronology) {
     assertEquals("eras", chronology.eras().getName());
     assertEquals("centuries", chronology.centuries().getName());
     assertEquals("years", chronology.years().getName());
     assertEquals("weekyears", chronology.weekyears().getName());
     assertEquals("months", chronology.months().getName());
     assertEquals("weeks", chronology.weeks().getName());
     assertEquals("days", chronology.days().getName());
     assertEquals("halfdays", chronology.halfdays().getName());
     assertEquals("hours", chronology.hours().getName());
     assertEquals("minutes", chronology.minutes().getName());
     assertEquals("seconds", chronology.seconds().getName());
     assertEquals("millis", chronology.millis().getName());
  }

public static @DataPoints BaseChronology[] classInstances = {
   IslamicChronology.getInstance(),
   EthiopicChronology.getInstance(),
   BuddhistChronology.getInstance(),
     GregorianChronology.getInstance(),
     CopticChronology.getInstance(),
     GJChronology.getInstance(),
     JulianChronology.getInstance(),
     ISOChronology.getInstance(),
     LenientChronology.getInstance(GregorianChronology.getInstance())};
```

Figure 1.2: JodaTime tests from Figure 1.1 written as a unified theory.

Chapter 2

Background and Definitions

The work in this thesis was done using the Java programming language and the JUnit unit testing framework, and the Soot static analysis framework [24]. This Chapter describes the existing patterns in Java unit testing and touches on relevant Soot specific terminology.

2.1 Unit Testing in Java

In Java, unit tests are typically written with the help of a testing framework. A typical unit test, as in Figure 2.1 contains the following components:

1. **Setup code:** Create test environment for the unit of code under test
2. **Test code:** Preparing an object for testing desired functionality
3. **Assertion Statements:** Verification of the prepared object's state
4. **Teardown code:** De-construct the test environment

The `setUp()` method is responsible for initializing the unit's state before individual tests are performed. It is the primary de-duplication mechanism used in unit testing.

The test itself is comprised of two parts. First is the construction of object state that involves invoking code under test. The second part of the test are the assertion statements that verify the object state(s) generated by the code under test.

```
public class TestLogger extends TestCase {

        Logger logger;
        protected void setUp()
        {
                logger = new Logger("test.log");
        }

        public void testDebugLevel()
        {
                logger.setLevel(DEBUG);
                logger.debug("This data is in the log");
                assertEquals(logger.getlogFile().getlineCount(),1);
        }

        protected void tearDown()
        {
                logger.logFile.truncate();
        }
}
```

Figure 2.1: Sample unit test for illustrating JUnit testing pattern.

The final component, the `tearDown()` method, returns the test environment to its pre-test state.

JUnit is an open source unit testing framework for Java. It provides mechanisms to easily construct and run unit tests. In version 3.x of JUnit these methods are contained in a class that extends JUnit's `TestCase` class.

The wide adoption of JUnit allows us to design our implementation specifically for this framework. We consider calls to methods declared in the `junit.Assert` class to be assertion statements, (referred to as *assertions* from now on.) We use assertions to identify data useful in comparing the similarity of tests since assertions are usually passed objects which have relevant properties to be verified. Effectively, we exclusively investigate JUnit assertions and related code for test clone detection.

2.2 Soot and Static Analysis

Soot is a compiler framework developed by the Sable Research Group at McGill University. It is used to build static analysis tools for Java source files or Java bytecode.

Since our approach gathers data using the new notion of Assertion Protocols (defined in the next Chapter), any intra-procedural static analysis framework for Java would be suitable for our work. However, some features of Soot made it a better fit for our purpose.

Soot analyses operate with the framework's own intermediate representation (IR), *jimple* of Java code which simplifies the code for the analysis. Jimple is a form of 3 address code, where each statement is broken down into multiple jimple instructions with 2 operands and an operator. Using simple 3 address code IR normalizes code fragments, for example,

```
z = x.foo().bar();
```
and
```
y = X.foo();
z = y.bar();
```

would be syntactically identical, with possibly different variable names since method chaining is not allowed in Jimple. A detailed guide to jimple has been published in [39], A jimple conversion of the test code in Figure 2.1 is provided in Figures 2.2, 2.3, 2.4.

In addition to Jimple, Soot provides a set of inbuilt 'helper' functions (like a use-def analysis and a class hierarchy extractor amongst others) which expedited our development

```
public class org.test.TestLogger extends junit.framework.TestCase
{
  protected void setUp()
  {
    org.test.TestLogger this;
    org.test.Logger temp$0;

    this := @this: org.test.TestLogger;
    temp$0 = new org.test.Logger;
    specialinvoke temp$0.<org.test.Logger: void <init> (java.lang.String)>("test.log");
    this.<org.test.TestLogger: org.test.Logger logger> = temp$0;
    return;
  }
```

Figure 2.2: Jimple 3-address code version of the setUp method.

process. The flexibility of Soot also allowed us to integrate thin slicing based data gathering techniques within the analysis with ease.

The rich feature set and research oriented nature of Soot makes it an ideal platform for future work. Soot supports dynamic analysis, inter-procedural analyses, symbolic execution and automatic refactoring amongst other possibly relevant functionality. Development using such an evolving platform makes taking new directions with our research easy.

2.3 Summary

While neither JUnit nor Soot are core requirements for implementing our clone detection technique, the ubiquity of JUnit and the extensibility of Soot make them ideal for demonstrating our work.

JUnit is widely used an is a primary testing framework for many open source projects. Thus making an implementation specific to JUnit is likely to benefit a large number of projects while being extensible to other imperative unit testing frameworks. Similarly, an extensible dataflow analysis framework with an active and stable developer community make Soot an ideal platform for current and future work in the field of program analysis based techniques.

In the next Chapter we discuss how we use Soot to implement a clone detector for tests written using JUnit.

```
public void testDebugLevel()
{
  org.test.TestLogger this;
  org.test.Logger temp$0, temp$2, temp$3;
  java.lang.String temp$1;
  org.test.lFile temp$4;
  int temp$5;

  this := @this: org.test.TestLogger;
  temp$0 = this.<org.test.TestLogger: org.test.Logger logger>;
  temp$1 = this.<org.test.TestLogger: java.lang.String DEBUG>;
  virtualinvoke temp$0.<org.test.Logger: void setLevel(java.lang.String)>(temp$1);
  temp$2 = this.<org.test.TestLogger: org.test.Logger logger>;
  virtualinvoke temp$2.<org.test.Logger: void debug(java.lang.String)>("This data is
      in the log");
  temp$3 = this.<org.test.TestLogger: org.test.Logger logger>;
  temp$4 = virtualinvoke temp$3.<org.test.Logger: org.test.lFile getlogFile()>();
  temp$5 = virtualinvoke temp$4.<org.test.lFile: int getlineCount()>();
  staticinvoke <org.test.TestLogger: void assertEquals(int,int)>(temp$5, 1);
  return;
}
```

Figure 2.3: Jimple 3-address code version of the test method.

```
  protected void tearDown()
  {
    org.test.TestLogger this;
    org.test.Logger temp$0;
    org.test.lFile temp$1;

    this := @this: org.test.TestLogger;
    temp$0 = this.<org.test.TestLogger: org.test.Logger logger>;
    temp$1 = temp$0.<org.test.Logger: org.test.lFile logFile>;
    virtualinvoke temp$1.<org.test.lFile: void truncate()>();
    return;
  }
}
```

Figure 2.4: Jimple 3-address code version of the tearDown method.

Chapter 3

Clone Detection Approach

Based on our empirical study, test code is usually simple and contains assertions comparing local variables to each other or to constant values. We can therefore, track histories of local variables in test methods and search for similarities between them to identify similar tests.

Figure 3.1 introduces our running example. It defines two classes, `List` and `Queue`, both of which inherit from a common abstract parent class `Collection`. The two subclasses each implement their parent's `insert()` and `length()` methods.

Since `Collection` is an abstract class, developers cannot test it directly. They must test it using its derived classes `List` and `Queue`. Using regular concrete unit tests (CUTs) forces developers to duplicate tests of the `length()` and `insert()` methods for both the `List` and `Queue` concrete classes. It would be feasible to construct a parallel test hierarchy to test common features at the `Collection` level if the level of behaviour of each child is known before hand. This is not always the cases since classes can implement or override methods declared in the parent.

Figure 3.2 shows two tests, both called `testInsert()`, which exercise the `List` and `Queue` `insert()` methods.

Such clones in tests increase test suite brittleness: code modifications can invalidate many test cases simultaneously. For example, if a developer changed the signature of `insert()` to accept strings instead of floats, both test methods would no longer compile. Furthermore, using regular CUTs forces developers to write new tests for each new subclass of `Collection`, resulting in more work as well as larger test suites.

We defined Assertion Protocols (APs) for the purpose of collecting data about variables at each line of code in order to detect cloned CUTs. The first step in our approach is to

```java
abstract class Collection {
  abstract void insert(float f);
  abstract int length();
}

class List extends Collection {
  // ... method implementations
}
class Queue extends Collection {
  // ... method implementations
}
```

Figure 3.1: Simple system under test: Class hierarchy with abstract collection and two implementations.

```java
void testInsert() {
  List list = new List();
  list.insert(3);
  int len = list.length();
  assertSame(1, len);
}
```

(a) Concrete Unit test for the List class.

```java
void testInsert() {
  Queue queue = new Queue();
  queue.insert(3);
  int len = queue.length();
  assertSame(1, len);
}
```

(b) Concrete Unit test for the Queue class.

Figure 3.2: Concrete Unit Tests (CUTs) for Collection class subtypes.

compute Assertion Protocols (APs) for local variables used in assertions. Assertion Protocols are devised such that similarity between APs indicates a higher likelihood of related code being a refactorable test clone.

Definition 1. *The* Assertion Protocol (or AP) *for a variable at a given line of code is a list of method signatures or constant types containing the declaration signature of each method invoked in order to compute the variable's value or the constant type is has been assigned.*

We chose Assertion Protocols as the basis for our technique because they capture facts exclusively about the properties developers test in a particular assertion. As a result, they can identify tests where only a subset of tested properties are similar rather than the entire test method. Assertion Protocols as we define them have not been used for static analysis previously and our implementation of the analysis to generate APs in Soot is a novel contribution.

The use of Assertion Protocols differentiates our work from traditional clone detection: because APs use def-use information, they are more resistant than clone detectors to textual code restructuring, differences in method call orders, and statement reordering. APs also help provide context for each of the clones reported to the developer. The above factors along with the clear connection to the code maintained by Assertion Protocols make them the ideal data source for our similarity analysis.

In the running example's case of the list `testInsert()` test method, we generate an Assertion Protocol for the `list` variable consisting of a single tuple: `[<List:List()>]`, indicating that this variable arose from the constructor call `List()`. We would also generate a Assertion Protocol for `len` which would be: `[<List:List()>, <Collection:length()>]` indicating the method calls in `len`'s history.

The queue `testInsert` method also generates Assertion Protocols for its local variables in the same way. Its `len` would instead have the Assertion Protocol `[<Queue:Queue()>; <Collection:length()>]`. We describe the process that leads to these APs in Section 3.1.

After generating the Assertion Protocols, we perform similarity analysis to compare each pair of JUnit assertions. After computing APs, we group together sets of assertions based on the similarities of the Assertion Protocols associated with each variable. A detailed description of this process is in Section 3.2.

The result of the first two phases is raw data on similarities between pairs of assertions in the various tests in the test suite. Because we aim to present method level candidates for refactoring to the developers, we consolidate our data to test level granularity in order to obtain candidate sets. We do this by consolidating similar assertions in code into test method level groups. In this final phase, we extract relevant method names from the sets and present them to the developer. We now describe each of the steps up to the final display phase in detail.

3.1 Assertion Protocol Generation

Figure 3.3 illustrates the rules for generating Assertion Protocols. At each assignment statement in the three-address intermediate representation used by Soot, we append a tuple or a set of tuples to the assertion protocol of the left-hand side.

Specifically, given an assignment statement `s: x = rhs`, we generate:

Assignment to x	**AP(x)**
Instantiation (`x = new X();`)	`[<X: X()>]`
Copy of local variable (`x = y;`)	`AP(y)`
Method invocation (`x = y.foo();`)	`AP(y).append( declaringSignature(foo))`
Static Method invocation (`x = staticFoo();`)	`[declaringSignature(foo)]`
Constant value (`x =` v`;`)	`[<Constant, typeOf(`v`)>]`

Figure 3.3: Rules used in Assertion Protocol Generation (at assignment statements).

In the static analysis multiple APs can arise due to branches in the code. Since static analysis is conservative, it computes an AP on each path and merges them when the flows join. Figure 3.4 contains a snippet of code where the variable `status` may have one of two distinct APs based on a branch in the code. When the branch merges, we assign *both* to `status`. The dataflow analysis steps are summarized in Table 3.4b.

For merging, the analysis associates both APs with the variable. Figure 3.4 demonstrates this with an example. Since we do not know what the method `login()` will return statically, the analysis handles each branch of the program independently. Assigning the appropriate AP to `status` in each branch block. When the control flow merges just before the assertion, both APs are assigned to `status`. The formal data flow analysis definition is in Appendix A.

In the case of our running example, we generate the APs for the variables as given in Figure 3.5. The APs for `Queue` would be generated similarly.

During the similarity analysis, when comparing two variables, we consider the maximum similarity score computed by comparing every pair of APs.

```
String status = new String();
if(login()==SUCCESS)
  status=LOG.successMessage();
else
  status=LOG.failureMessage();
assertNotNull(status);
```

(a) Sample listing containing branching code.

Statement	**APs**
`String status = new String();`	`AP(status)=[<String:String()>]`
`if(login()==SUCCESS)`	**Branch1**
`status=LOG.successMessage();`	`AP(status)=[<LOG:successMessage()>]`
`else`	**Branch1**
`status=LOG.failureMessage();`	`AP(status)=[<LOG:failureMessage()>]`
Branch Merge	`AP(status)={[<LOG:failureMessage()>], [<LOG:failureMessage()>]}`
`assertNotNull(status)`	`AP(status)={[<LOG:failureMessage()>], [<LOG:failureMessage()>]}`

(b) APs at each line of code as generated by the dataflow analysis.

Figure 3.4: An example of branch handling during AP Generation.

3.2 Similarity Analysis

The similarity analysis identifies similar test assertions by generating similarity scores for each pair of assertions. Algorithm 1 summarizes our heuristic for assertion similarity computation.

We score a pair of Assertion Protocols by counting the number of common elements that exist between them. If two APs are of the same length we check if they contain the same entries in the same positions in the list. For each matching declaration, we increase the score incrementally within the range $[0, 1]$.

For a pair of assertions we compute the similarity score by adding all the similarity scores for assertion parameters' APs along with additional factors and normalizing to $[0, 1]$.

The heuristic is based on the following criteria:

Assertion Type Unit tests execute some code from the system, and compare the actual result of the execution to an expected value. In JUnit tests, the comparison uses JUnit-provided assertions. Examples of these assertion types include `assertNotEqual`, `assertTrue`, and `assertSame`. Our heuristics assumes that it is unlikely for tests using different assertions to be clones and assigns a similarity score of 0 to assertions with different assertion types.

Assertion Protocol Similarity The similarity score between 2 assertions is the core of our algorithm. For assertions a_x and a_y, corresponding assertion parameters (a_x^i, a_y^i), we compute the number of pairs in the Assertion Protocol that match and divide this count by the total length of the Assertion Protocol giving a result in $[0, 1]$ for each assertion parameter. We match a pair of entries based on the class hierarchy of the declaring types of the two calls. Two calls match if they are declared or defined by a common ancestor in the System's type Hierarchy.

Assertion Constants Many test assertions compare the actual result to a constant expected result. Note that our heuristic handles this case by encoding an Assertion Protocol for constants containing the type of the constant.

Containing Method Name Our preliminary inspections found that assertions belonging to identically-named methods (in different tests) indicate duplication. We therefore assign 1 similarity point when evaluating assertions belonging to methods with the same name while computing the average similarity score.

In the case of `List` and `Queue` we compare the APs of `len` and `1` for each of the tests and compute the similarity score.

The APs for each variable named `len` are:

- While testing `List`: `[<List: List()>,<Collection: length()>`
- While testing `Queue`: `[<Queue: Queue()>,<Collection: length()>`

The similarity score for these 2 APs is $\mathbf{(0 + 1)/2 = 0.5}$ given that one out of 2 corresponding AP entries match. The variable name does not factor into the similarity score and can be different without affecting similarity scores.

In the case of the second assertion parameter, `1` in each of the tests, the constant is of the same type, `Integer` and so the similarity score is **1**.

Since the test method name is `testInsert` in both cases, we have to add an additional point of comparison for matching containing method names. Giving another **1** point to the similarity score.

The final score is an average of the assertion protocols and the extra point for method name similarity resulting in a final similarity score of $\mathbf{(0.5 + 1 + 1)/3 = 0.833}$.

The details of the similarity analysis are given in Algorithm 1.

3.3 Result Consolidation

After computing the similarity between pairs of assertions in different tests, we group assertions based on which test method they belong to in an effort to identify test methods with similar behaviour and associated assertions.

We use an undirected graph with assertions as nodes and edges between pairs of assertions whose similarity is within a tunable range range; for this thesis, we report results with a threshold of $[0.6, 1]$. Different thresholds give different results; decreasing the threshold produces a superset of the results with potentially lower precision. We store the similarity scores between assertions as the corresponding edge weights.

```
function **similar** (assert1, assert2)
  similarity ← 0
  if(assert1.type() ≠ assert2.type())
    return 0
  for(variable1, variable2 ← assert1.parameters, assert2.parameters)
    similarity += likenessScore(variable2, variable2)
  if(assert1.testMethodName != assert2.testMethodName)
    if (similarity / parameterCount in [range])
      return true
  else
    if (similarity+1 / parameterCount+1 in [range])
      return true
  return false

function likenessScore (variable1, variable2)
  score ← 0
  for(APForV1, APForV2 ← variable1.assertionProtocols,
                        variable2.assertionProtocols)
    if( APForV1.length ≠ APForV2.length)
      continue;
    APScore ← 0
    for(method1, method2 ← APForV1, APForV2)
      if(Soot.existsSharedHierarchy(method1, method2))
        APScore ← APScore + 1
    APScore ← APScore / AP.length
    if(APScore > score)
      score ← APScore
  return score
```

Algorithm 1: Variable AP similarity score calculation formula

We search the graph for isolated, connected sub-graphs. Such components are likely to contain good candidates for refactoring.

To consolidate our results, we take these mutually disjoint connected sub-graphs and attempt to reduce each one to the test level relations between the assertions.

We explore the graph considering only edges with the same similarity score at one time. since edges with the same weight are empirically indicative of related Assertion Protocols. We compact these edges, grouping assertions based on which test method they belong to. The new compacted graph has test methods as nodes and similar test methods connected via edges. This connected graph of related methods is representative of a test level clone and is displayed as a set with supporting assertion similarities in the final display for the developer. In our running example, there are only two test methods and so drawing the graph is not very interesting. There will be 2 nodes with an edge between them with a weight of **0.833**. Consolidating the graph will not result in any change. However, if there were multiple assertions which matched each other, we would see them reduced to a single pair of nodes which would represent the test methods.

Figure 3.6 presents a typical set of relations between assertions. Each vertex, a_1 through a_8, represents an assertion from some test. Assertions a_1 through a_4 come from four different (but related) types of FIFO queues in the Apache Commons Collections library.

3.4 Final Display

Our tool presents developers with a set of test methods as recommendations. The final stage of our approach links the connected components of test methods back to their original source code.

To find candidate methods for refactoring, we display the unique method names associated with assertions in each graph component as one set. Our output also includes hyperlinks to the assertions which were used as a basis for making the recommendation, it can be used to jump to individual assertions in the test code using Eclipse.

Since the clones detected by our approach may not be textually similar, developers can rely on the assertions we present to decide whether or not the 2 related test assertions are testing the same conceptual property and if that part of the test should be refactored to remove the duplication.

The final display for our running example is given in Figure 3.7, a real world example of final display from the Jodatime test suite is given in Appendix C

When presenting the sets of methods, we prioritize them based on the similarity scores of the edges that exist in the component. Each component appears exactly once in the output but individual methods may exist in multiple components with different edge weights.

3.5 Conclusion of running example

To end our running example, a developer can exhaustively test both `insert()` and `length()` for every applicable child of `Collection` using JUnit theories. Figure 3.8 presents the JUnit theory based test. Theories use the annotation `@Datapoints` to define data for testing against a theory annotated with the `@RunWith(Theory)` annotation. The test verifies consistent behaviour of `insert()` and `length()` in subtypes of `Collection`, without duplicating the test cases as in Figure 3.2. It is easy to see that a new subclasses of Collection can be tested with relative ease. While the savings in lines of code is not appreciable in this case, it would clearly be beneficial for non-trivial tests as seen in the Introduction (Chapter 1).

3.6 Summary

This chapter explains the technique we developed to generate a similarity metric based on the notion of Assertion Protocols. It describes the static analysis we implemented to generate the APs, the method to calculate the similarity metric and the final display format we used. In the next Chapter, we discuss the results of our empirical study where we used our tool to detect test clones in 10 real world test suites. real world

Statement	**APs**
`List list = new List();`	`AP(list)=[<List:List()>]`
`list.insert(3);`	`AP(list)=[<List:List()>]`
`int len = list.length();`	`AP(list)=[<List:List()>],` `AP(len)=[<List:List()>,<Collection:length()>]`
`assertSame(1, len);`	`AP(list)=[<List:List()>],` `AP(len)=[<List:List()>,<Collection:length()>,` `AP(1) = [<Integer>]`

(a) Assertion Protocols Generated for List class' testInsert() method

Statement	**APs**
`Queue queue = new Queue();`	`AP(list)=[<Queue:Queue()>]`
`queue.insert(3);`	`AP(queue)=[<Queue:Queue()>]`
`int len = queue.length();`	`AP(queue)=[<Queue:Queue()>],` `AP(len)=[<Queue:Queue()>,<Collection:length()>]`
`assertSame(1, len);`	`AP(queue)=[<Queue:Queue()>],` `AP(len)=[<Queue:Queue()>,<Collection:length()>,` `AP(1) = [<Integer>]`

(b) Assertion Protocols Generated for Queue class' testInsert() method

Figure 3.5: Assertion Protocols Generated for list test in running example (Figure 3.2.

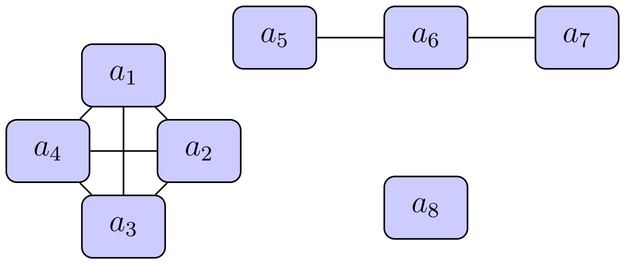

Figure 3.6: Graphical view of assertions (nodes) and their similarity (edges, which denote assertions whose similarity exceeds the threshold).

```
Pattern followed:
Clone Set:
<Method_Name>
<Set of Similar_Assertions>

Clone Set:
TestQueue.void testInsert():
(TestQueue.java:5)
TestList.void testInsert():
(TestList.java:5)
```

Figure 3.7: Final results displayed for the running example by the implementation.

```
@RunWith(Theory.class)
void testCollectionInsertLength(Collection c)
{
    c.insert(1);
    int len = c.length()
    assertSame(len,1);
}
@Datapoints
Collection collections[] = [new List(),
                            new Queue()];
```

Figure 3.8: Expected theory based tests for the Collection class' children.

Chapter 4

Results of Empirical Study

This chapter describes the findings of our empirical study conducted To validate our clone detection approach, we performed an empirical study, applying our tool to a collection of 10 Java-based open-source real-world. We chose systems which have significant JUnit based test suites. Table 4.1 lists our subject systems, their sizes, the sizes of their test suites[1], the version of the system we analyzed, and running times for our analysis.

System	**LOC**	**Test LOC**	**Version**	**Analysis Runtime**
Apache POI	85,379	57,536	1389914	6m50s
Commons Collections	26,495	29,559	645800	4m31s
Google-Visualization	8,027	9,382	70	6m51s
HSQLDB	160,665	19,136	5076	4m03s
jDom	18,916	15,428	a9a2f863c5	29s
jFreeChart	93,998	51,710	3561093	9m07s
jGrapht	11,342	6,225	a8056d6aaf	50s
jMeter	90,504	14,511	1431816	1m24s
JodaTime	26,881	51,497	1613	9m33s
Weka	246,828	13,030	9415	1m39s
TOTAL	769,035	268,014		

Table 4.1: Benchmark systems.

[1]LOC data generated using David A. Wheeler's SLOCCount

4.1 Experimental Results

We manually assessed ten random refactoring recommendations for each test suite and report the expected reduction in test case count based on the primary author's judgement of the recommendations Of the random sample we chose the recommendations. In this section, we discuss the nature of the recommendations and the patterns identified by our approach.

System	**Test Methods**	**Clone Sets**	**Matched Methods**	**Methods /Set**	**Matched Methods/Suite**
Apache POI	4743	2455	1832	23	39%
Commons Collections	2653	979	660	20	25%
Google-Visualization	485	312	233	7	48%
HSQLDB	1826	482	433	22	24%
jDom	371	155	134	6	36%
jFreeChart	3556	1488	1290	200	36%
jGrapht	297	105	97	4	33%
jMeter	982	523	453	8	46%
JodaTime	4746	4549	2592	71	55%
Weka	1519	702	307	41	20%
MEAN	2117.8	1175	803.1	40.2	36%

Table 4.2: Test clones reported by our analysis.

Figure 4.2 summarizes our results quantitatively. The `Test Methods` column lists the number of test methods in the system—specifically, the number of methods contained within classes that extended the `TestCase` class from JUnit. The `Clone Sets` column reports the number of clones we discovered, while `Matched Methods` reports the total number of unique methods that appear across all clones. The next column reports the average size of each clone set. The column `Matched Method %` reports the percentage of test methods that were present in at least one clone set.

Based on our tool, a significant fraction of the test suite contains test clones. However, the results are highly dependent on architecture and code style and do not necessarily report code suitable for retrofitting in every case. Intuitively, object-oriented systems with deep class hierarchies have many clones than systems with flat class hierarchies yield fewer results owing to the lack of shared architectural properties.

The testing style of each test suite also greatly influenced our results. Implicit testing results in clones that on investigation are not functionally similar. Behavioural tests, which verify if appropriate methods were called, are not analysed effectively by our technique. This is caused by the assertions not being used as expected and verifying if methods return successfully rather than object state.

In case a suite was already using techniques to avoid clones, our analysis had fewer points of comparison and poor results. Since there were fewer assertions and they were often located in different methods from test code, the intra-procedural nature of our analysis did not associate the test code with the assertions, even were clones could be observed when reviewing code manually.

We now discuss common themes and notable features of the specific test suites that affected our analysis results.

4.2 General Trends

While reviewing the use of assertions in our test suites, we found that systems used a variety of techniques to consolidate common tests. While these techniques may use different syntaxes, their intents appear to be similar to each other. We found the first two approaches used in the wild by our test systems:

Abstract test classes Using abstract test classes that implement tests required for an set of classes effectively reduces the total LOC and improves coverage while simplifying code. This technique uses no extra tooling relying solely on Java's inheritance; This makes it simple to use, understand and maintain. The `Weka` project leverages this technique and has the smallest ratio of test code to production code amongst all the systems in our benchmark.

Helper functions Another technique which does not require additional infrastructure is the use of helper functions to validate constraints on a variety of objects. This approach is used in the Apache Commons-Collections library and Weka. It is used to a lesser extent in other test suites but does not see any use in JodaTime.

Parametrized tests and theories Parametrized tests or theories apply a test to multiple objects or types. These are explicitly designed to improve test coverage while avoiding code bloat. Therefore they are ideal for clone removal when their application is feasible.

4.3 Developer impressions

To evaluate the quality of our recommendations, we examined 10 random results from systems picked to be representative of high, medium and low priority candidates. It was easy to decide whether a recommendation was useful or not: in the cases where our tool provided an unusable recommendation, it was easy for the authors (who were unfamiliar with the subject codebases) to detect that the recommendation was not applicable and discard it. If the clone set was a valid candidate for refactoring, it was usually straightforward to formulate refactorings. The greatest difficulty in applying the recommendations was in deciding what mechanism to use for refactoring; a higher-level understanding of the code structure would and code style allow a developer to choose the best technique for refactoring the tests.

Explanations of theories often start by showing how they are applicable to `hashCode()` and `equals()` methods[2]. We indeed found that, in 7 of our 10 benchmark test suites, we detected that `testHashCode()` methods could be refactorized. This result suggested the potential for our technique to make valid recommendations in other less-clear cases as well. We also identified refactorable `testEquals()` methods in a number of our suites, including Apache Commons Collections and JodaTime.

4.3.1 JodaTime

JodaTime has a comprehensive test suite which is particularly amenable to refactoring; It tests multiple different classes which are siblings and are composed of the same types. As a result, our system reports 54.6% of its test methods as clones.

The JodaTime suite is an outlier in terms of test suite size compared to system size: the test suite is nearly twice as large (in LOC) as the actual system. JodaTime's tests are particularly suitable for our approach as they use fine-grained, explicit, assertion-driven verification. Even though the tests usually operate on different types, they are actually verifying the same property, which is specified in the common abstract superclass.

We also found that, while JodaTime contains many test clones, it also contains many "locally-parametrizable" tests, which we explicitly do not attempt to identify. Such tests verify the behaviour of a *single* object on multiple inputs. Our goal is, instead, to identify tests of multiple objects, on the widest range of inputs available. We found that many of

[2]As an example: `http://www.smartics.de/archives/2060`.

the invalid recommendations for JodaTime were actually (somewhat complicated) locally-parametrizable tests. Our tool identifies the tests as belonging to a single component and hence amenable to refactoring. Refactoring also eliminates inconsistent code styles and prevents omission of assertions in the cloned tests.

```
// In TestPredicatedSet.java
public void testIllegalAdd() {
  Set set = makeTestSet<String>();
  Integer i = new Integer(3);
  try {
    set.add(i);
    fail("Int must fail string predicate.");
  } catch (IllegalArgumentException e) {
    // expected
  }
  assertTrue("Not added", !set.contains(i));
}
```

```
// In TestPredicatedBag.java
public void testIllegalAdd() {
  Bag bag = makeTestBag();
  Integer i = new Integer(3);
  try {
    bag.add(i);
    fail("Int must fail string predicate.");
  } catch (IllegalArgumentException e) {
    // expected
  }
  assertTrue("Not added", !bag.contains(i));
}
```

Figure 4.1: Shared behaviour testing in Commons Collections.

4.3.2 Apache Commons-Collections

As seen in our motivating example in Section 1, test suites for collections are a likely source of clones. Our tool identified 38.6% clones.

Shared interface behaviour accounts for many of the findings in the suite. For example, in Figure 4.1, `Integers` must not be added to a `Set<String>` or a `Bag` returned from `makeTestBag()` (which happens to create a `Bag` constrained to only contain `Strings`). The tests are nearly identical and verify an implicit behavioural rule, hidden in the implementation.

An additional benefit of refactoring in collections (and other systems) would be that every collections' behaviour upon illegal operations must be consistent since there is a single test that validates the behaviour. The API consistency is therefore implicitly enforced in the tests where it may have been missing.

4.3.3 JFreeChart

Our tool identified 36.2% cloning out in JFreeChart. It is more challenging to refactor the tests identified here compared to results from Apache Commons-Collections: understanding whether tests are semantically close enough to refactor requires more domain understanding than in previous tests, and refactoring these tests may hamper readability. For example, in Figure 4.2, tests had very similar assertions; however, the first test used a 3-dimensional `DefaultXYZDataset` while the second test used a 2-dimensional `DefaultXYDataset`. The tests therefore verified similar but in some ways, distinctly different properties. We felt that the slight variations are difficult to express concisely without hurting readability. Our technique still correctly identified clones, however it is not clear if refactoring in every case would improve the test suite.

```
public void testAddSeries() {
  DefaultXYZDataset d = new DefaultXYZDataset();
  d.addSeries("S1", new double[][] {{1.0}, {2.0}, {3.0}});
  d.addSeries("S1", new double[][] {{11.0}, {12.0}, {13.0}});
  assertEquals(1, d.getSeriesCount());
  assertEquals(12.0, d.getYValue(0, 0), EPSILON);
}

public void testAddSeries() {
  DefaultXYDataset d = new DefaultXYDataset();
  d.addSeries("S1", new double[][] {{1.0}, {2.0}});
  d.addSeries("S1", new double[][] {{11.0}, {12.0}});
  assertEquals(1, d.getSeriesCount());
  assertEquals(12.0, d.getYValue(0, 0), EPSILON);
}
```

Figure 4.2: Two similar testAddSeries() implementations in jFreeChart.

4.3.4 HSQLDB

Our tool identifies very few refactorable tests for the HSQLDB suite at 23.7%; these tests often verify that operations on a reader object must be unsuccessful prior to an `open` call on that object. Or similar constraints.

During our initial investigation we found that many tests in the suite simply verified return codes from SQL functions, rather than explicitly testing behaviour or underlying state. Also, the HSQLDB codebase contains shallow inheritance hierarchies which do not provide rich assertion protocols.

A typical HSQLDB test case (Figure 4.3) calls a number of side-effecting method calls on a `PreparedStatement` object. Unfortunately, the tests then verifies the success of the `executeUpdate()` call by checking that the return value is 1; The test never verifies that the update actually produced the expected change in state. Due to this unique testing strategy, our technique finds many clones which test unrelated properties.

```
public void testSetNull() throws Exception {
  PreparedStatement stmt = updateColumnWhere("c_integer", "id");
  stmt.setNull(1, Types.INTEGER);
  stmt.setInt(2, 1);
  assertEquals(1, stmt.executeUpdate());
}
```

Figure 4.3: An implicit unit test in HSQLDB.

4.3.5 JGraphT

Our tool identified 32.6% cloning in the JGraphT test suite. However, given the relatively small size of the test suite, we were not able to find many useful clone sets with our tool. We noticed some similarities compared to the case of HSQLDB in that many times, the assertions verify easy to verify attributes, (in this case, vertex count) and not the actual structure of graphs being generated.

4.3.6 jDom

Most of the test code in jDom constructs XML Document Object Models and then validates them at the end of the test method against contents in the file system. However, in smaller tests we identified many snippets of code that could be placed in helper functions. These snippets typically involved setting up basic XML structures and sanity testing.

4.3.7 Weka

Our tool returned he lowest match rate of 20.2% for Weka. We explored the test suite and found that it was also an outlier: the Weka developers already use inheritance aggressively in the test suite. Additionally, they use very few explicit assertions (only 119 in over 3000 tests). Essentially, they have already optimized the tests using object oriented design and test class inheritance. This results in an especially compact test suite: in our benchmark set, it has the fewest lines of test code to benchmark code of any of our benchmarks. We believe that this result indicates that some form of parametrization of test suites is valuable and that tool support for it would be helpful to test writers.

4.4 Summary

This Chapter discussed the results of our empirical study on 10 JUnit based benchmark test suites. We found that while there are many clones in test code, the decision to refactor them must be made based on code style on a suite-by-suite basis.

In certain suites, the standard practice involved test code reuse and detected clones were not amenable to refactoring. In other suites, implicit testing without assertion based verification of object state lack fine grain APs in assertions, which are not useful for reasoning about the retrofitting process.

However, in suites which test deep hierarchies and limited code reuse, our approach does detect a large number of refactorable clones and is a beneficial tool for developers to use.

Chapter 5

Discussion

Overall, we found that our tool produced refactoring recommendations for almost all of our benchmark suites, and that it worked particularly well on JodaTime. We continue by discussing some of our broader, cross-benchmark, qualitative findings.

When our tool produced any recommendations, it tended to produce high-quality ones: our recommendations contained very few false positives, and it was trivial for non-experts to identify false positives without too much effort in most cases.

Retrofitting based on our recommendations was sometimes, but not always, straightforward. We believe that refactoring of tests is not always feasible. For instance, the parameters to be passed while parametrizing certain cases needed to be class types—test objects were fetched using static methods. This particular behaviour is not directly supported by Java and cannot be trivially refactored.

5.1 Scope of Our Analysis

Our tool yields better results with certain test suites than others. In preliminary work, we experimented with running our tool on automatically-generated test suites (in particular, the test suite for the Google Guava libraries). We did not detect refactoring opportunities in such suites, as the suites are designed to maximize test coverage while minimizing suite size. Suite size minimization implies that refactoring is unlikely to help.

Our analysis also does not find significant valid refactoring opportunities in already-factored test suites like that of Weka.

Some tests did not use explicit assertions; we saw some such tests in every suite except for JodaTime. Instead, these tests look for explicit failures, e.g. exceptions, or implicit self-reported success checks (as in HSQLDB). Such tests are difficult to reason about, either manually or automatically.

5.2 Retrofitting vs. Early Optimization

We found that the choice of test style affects whether the test suite will be easy or hard to theorize or parametrize. For example, JodaTime uses fine-grained tests which can be easily rewritten as parametrized tests, while JFreeChart's similarly-complex tests were much more difficult to evaluate for retrofitting.

In the initial stages of project development, when projects are still small, there may not be much of an incentive to use complex testing techniques; example-based concrete unit tests (if any) would be more practical. Over time, the system may evolve into one that is suitable for theory-based testing; however, the incremental growth of the system may make it difficult to recognize this fact. Thummalapenta et al. propose a cost benefit analysis to decide whether theorizing a test suite would be feasible and would have favorable results [37]. Our tool provides a low-cost way to help developers decide whether they should retrofit their tests yet, and to guide them if they proceed to do so.

5.3 Comparison with Clone Detectors

It is evident that we need clone detection in test code based on the results presented above. We believe that clone detection in tests can be used to reduce test duplication and is a beneficial exercise.

Based on a comparison of clones detected by our algorithm and those of the best in class clone detector Bauhaus [15], specialized tools to detect clones in test code can be beneficial. We compared 10 random samples from each test suite with results from Bauhaus matching with token length threshold set of 50. Bauhaus produced more clone sets, however, our approach lead to results which had semantic data used to define the similarity. We believe that our empirically-generated algorithm produced more useful results within the limited scope of test code. We reviewed the results and found this limited subset of results with semantic information easier to reason about and refactor compared to Bauhaus' clone detection results.

Clone detectors do not typically provide the basis for clone detection along with the results. Our display expresses the features responsible for identifying code clones at the Java instruction level. This information is useful to developers and in the future, automated tools that may use this knowledge to evaluate: (a) the validity of our results and (b) the best refactoring approach to take.

Our analysis can detect similarity in tests which go beyond simple clone detection; for instance, we are insensitive to the order in which assertion parameters are computed, since we construct Assertion Protocols independently for each parameter. However, in practice, many of the recommendations from our tool were simple text clones, where the cloned code is nearly identical in both cases. Such clones are quite easy to refactor.

We believe that these easy clones will help motivate developers to use our tool. Furthermore, since we include more just simple clone analysis, we believe that our technique has the potential to identify more opportunities. In our experience with the tool, we had to decide whether refactoring the tests was worth the effort and risks involved. This was not always a straight forward decision (especially for more complex clones) since the clarity of what a test is intended to verify may be obscured. We think that this judgement call would be easier for developers who are more familiar with a project, even if they are unfamiliar with the specific tests to be refactored.

5.4 Result Quality

The quality of results generated by our algorithm is a subjective property and will always be biased based on the developer reviewing the results. We attempted to use a conservative criteria for reviewing the results by mimicking real developer behaviour: if a clear refactoring plan was not apparent within 2 minutes of looking at the code, we deemed the result poor in quality. The use of automated refactoring tools that leverage such semantic data would allow us to measure the quality of results more quantitatively but we did not find any existing tools suitable for this task.

5.5 Threats to Validity

The external validity of our system whether or not it would generalize to more systems, is limited by the number of systems we applied our approach to. While 10 systems were investigated, all were Java programs, none of them were industrial, and none exceeded 250

KLOC. It is also possible that our benchmark suite, though varied, did not capture other test code styles or approaches. Other test suites may utilize parametrization, theories and various other techniques to reduce clones. Since the recall of our approach is dependent on various properties of test code, we cannot conjecture about the results in those cases but the basis for detecting clones should still be valid. The construct validity of our results—whether or not we are measuring what we hope to measure, suffers from the fact that one author manually evaluated the quality of our returned results using a single scale. However, we believe that we have gained an appropriate understanding of the systems we investigated and have given an accurate assessment of our technique's performance.

5.6 Limitations

While using Assertion Protocols is effective, the similarity analysis relies on the size of the assertion protocols and the class hierarchies of the systems being analyzed. Therefore, in systems where class hierarchies are shallow our approach gives poor results. Further, it assumes that class hierarchies share common behaviour. The analysis provides unrefactorable results where inheritance does not imply shared behaviour, as was the case in the JFreeChart system we investigated.

Implicit rather than explicit property verification can also lead to similarly un-refactorable results. When tests validate if certain methods executed successfully rather than verifying underlying state, that fact is not captured in the related APs. HSQLDB was a system where the tool produced many results related to database update method where the tests were using radically different SQL query strings.

More specific treatment of constants would address the issue encountered in HSQLDB but would weaken our ability to detect tests which can benefit from more powerful refactorization techniques like theories. A more nuanced approach to constant matching may alleviate this shortcoming in the future.

During consolidation of the graph, we have to deal with the variable sizes of the components generated by the similarity analysis. If the component size is too large, the Final Display will list too many facts and methods to be refactored, only some of which may be of interest. Tuning the threshold parameters and considering only like weighted edges at once mitigates this issue but it is not clear how much that also impacts the recall rate and validity of the results.

Addressing these limitations are also matters of future work, which is discussed in the next section.

5.7 Future work

Based on insights gained during the implementation and evaluation of our technique, we found some “low hanging fruit” which would improve our tool without significant redesign. First, we wish to understand the equivalence of assertions like `assertFalse(fact)` and `assertTrue(!fact)` or assertions with optional parameters. This will give us opportunities to compare tests which we cannot match with our current implementation.

We can also look at non-assertion statements as points of interest when comparing 2 chunks of code. Looking at logging API calls for example, we can generalize our approach and attempt to discover clones in non-test code as well. The variability of logging frequency will be an added challenge we will have to address.

A more exhaustive method for computing assertion protocols could utilize inter-procedural analysis instead of the intra-procedural version we currently employ. This could capture side effecting calls which we do not address in this work. Another significant benefit would be the ability to address mock based testing, an alternate testing technique, which is not covered explicitly as part of this work.

Augmenting our static analysis using a dynamic trace as additional information or input from existing clone detectors will also yield similar benefits. Clone detector output can be leveraged to select interesting method calls for similarity matching as well.

Apart from the Assertion Protocol generation, we may also improve the similarity matching system by further considering the Assertion Protocols of variables passed as parameters in non-assertion method calls. In the current algorithm, this would be extremely compute and memory intensive and would likely not improve our results significantly compared to other approaches mentioned above. However, we believe that it may be useful for partitioning large clone sets, making them easier to refactor.

Finally, we would like to run a study to understand the usefulness of our approach for real developers and systems. We would like to design an Eclipse plug-in that streamlines our tool and makes it easy to use with any system. This will allow us to evaluate the utility in a variety of use cases. Closer integration with Eclipse could also come in the form of a refactoring extension that automatically applies improvements based on the findings of our analysis.

5.8 Summary

In this Chapter we presented the inferences we have drawn based on the data presented in Chapter 4. We also highlighted the limitations of our technique and threats to validity. Finally we discussed possible future work based on this research. In the next Chapter we present existing work related to our field and scope.

Chapter 6

Related Work

In this chapter we discuss areas of related work. Our initial motivation was found in papers that discussed new unit testing approaches which reduce test code redundancy while improving test quality. We describe these techniques in Section 6.1.

These new approaches are very interesting, but developers are unlikely to manually changing working test code to use them. Understanding different approaches to testing is critical in developing techniques to automatically detect where they are applicable and automate the retrofitting process.

The first phase of the retrofitting process is detection of retrofitting opportunities. The clone detection community focuses on identifying snippets of code which are likely copy-paste clones with some modifications. We attempted to use clone detection approaches on test code to see if more specialized techniques were required for clone detection in test code. An overview of clone detection in general code is given in Section 6.2.

Clone detection should lead to clone elimination in case the clones are undesirable. Automatic refactoring was thus the next logical step in our search for existing technologies, having reviewed the existing work, we found that the focus of test refactoring is to extend existing functionality based on automated techniques. We did not find any work which focused directly on our problem of refactoring manually written tests to use more modern testing techniques. We discuss automatic refactoring tools in Section 6.3.

As our work became more concrete, we discovered similarities between our technique and that of program slicing. Our work on Assertion protocol generation is in effect a spiritual reincarnation of the program slicing paradigm but chooses to preserve facts about the code slices rather than attempt to reason about the code slices themselves. We compare our work to program slicing in Section 6.4.

6.1 Alternative Testing Techniques

Saff, who originally proposed theory-based testing [33], conducted a case study which found that theory based testing could be applied easily to existing test suites [34]. Our work aims to provide tool support for adding theories.

Tillman and Schulte first proposed the idea of automatically parametrizing tests in [38], where they addressed parametrization within the context of one concern, using symbolic execution. Fraser and Zeller showed that exploratory data point generation tools can further improve test coverage without additional developer effort [12]. Zhang et al. proposed an automated approach to test generation in [44] which used a combination of static and dynamic analyzes to generate tests that extended coverage.

6.2 Code Clone Detection

Clone detection is the process of finding pieces of code that are similar based on a definition of similar which is suitable for the task. The similarity can be based on and criteria that is considered pertinent. The standard classification for clones however is limited to the source code, syntax and functionality as follows:

- **Type I:** Code that is identical except white spaces and comments;
- **Type II:** Type I clones with differences in identifier names but identical structure;
- **Type III:** Type II clones with minor changes in the structure;
- **Type IV:** Code that is syntactically and structurally different but has the same functionality;

Type I clones can involve method calls which have the same names but different, unrelated declarations which can be completely unrelated. Our approach does not detect such clones. However, such clones are unlikely to be suitable for retrofitting given that they test unrelated code.

Our tool detects Type II clones with the same scope and limitations as Type I clones because our approach does not depend on the variable names in computing similarity.

Type III clones are identified when there are changes within Type I/II clones that within some threshold. Since we use APs as a metric for similarity and not source code our approach is immune to unrelated source code preventing clone detection. Apart from non-reliance on source code, APs can be dissimilar within the tunable range and still be identified as clones, resulting in successful Type III clone detection.

Type IV clones may have no syntactic similarity and are not detected by our technique directly since our approach is dependent on a non-trivial fraction on the code being a clone of type I, II or III.

Clone detectors analyse source code and process to find similarities between code blocks. They accomplish this by first converting source code into an intermediate representation that can be processed for similarity based on a variety of techniques. The similarity can be based on a variety of criteria which influence the intermediate representation. They can be broadly classified as follows:

Text Based: Use the raw source code without significant normalization. A text based as in [17] can attempt to simply match pairs of contiguous strings using an efficient string comparison technique. Alternatively, more complex hybrid techniques can normalize and filter code to detect Type II and III clones as in [31] and [25].

Token Based: Use a normalized intermediate presentation, produced by a lexer which tokenizes source code. Token based analysis can be used to detect Type I and II clones. Type III clones can also be detected using a Token based similarity metric by identifying Type I or II clones which have differences that are below a (usually tunable) threshold. CCFinder is an example of a Token based approach [19]. Array based matching techniques also exist as seen in [4].

Tree Based: Tree based approaches attempt to find similarities based on a tree based representation of the code. A tree structure is used by CCFinder mentioned above in a hybrid approach to detect Type III clones. Tree based detection mechanisms often use the Abstract Syntax Tree generated for the code and identify similar suffix trees, which represent cloned code [23][5].

Metric Based: Metric based approaches do not compare code or some IR directly. Instead they define abstract properties based on the code which are compared for similarity. In [10] Davey et al. use neural networks to detect similarities between code based on facts such as frequencies of keywords and line indentation amongst others. Merlo et al. use properties of classes in object oriented code to generate metrics which can be used to model cloning in [27]

Graph Based: Graph based techniques analyse program dependency graphs or control flow graphs generated for the source code in order to detect similarities. Such approaches rely on static analysis tools to detect clones. This approach converts the problem into that of identifying isomorphic subgraphs, which is NP-complete. There are however simplifications and approximation methods which make the problem tractable. For example, Liu et al. use statistical filters to reduce the search space for isomorphism significantly in CPLAG [26]. The process of detection has also been reduced to a tree matching problem by Gabel et al. in [13].

Our approach is a hybrid that has many similarities with the program slicing technique used by Komondoor and Horwitz [22]. We use a graph-based technique to generate variable APs. These APs are then used to calculate a similarity metric for comparison of code at the test method level. The program slicing used by Komondoor and Horwitz was applied at the code level. However, our approach is not dependent on slicing explicitly, instead, the rules used to generate APs implicitly ignore non-assignment statements in code. We further partition the program by comparing similarity metrics of only variables used in a specific pair of assertions at one time.

Juergens et al. discuss how the evolution of clones often leads to bugs in code [18]. There is contention on this issue, Rahman et al. have found that clones are not inherently bad as characterised and may often result in less buggy or easier to fix code [29]. The usefulness of refactoring a clone is not always clear. Wang and Godfrey find that syntactic similarities between clones in the Linux kernel often indicate deeper architectural commonality and shared design patterns [41]. We identify such clones as good candidates for test refactoring.

However, Wang and Godfrey are also discuss the need for better contextual clone management techniques [40] providing [11] as an example of providing developers with contextual information within the development environment for developers to make more informed decisions. In our tool, we attempt to provide more contextual information for developers to judge if a clone is suitable for retrofitting and removal. This is not always clear from existing clone detection reports as suggested in ˜citeCordy:2003:CRP:851042.857051

6.3 Refactoring

The problem of discovering invariants for refactoring has been previously investigated for non-test code by Kataoka et al. [20]. The authors used an invariant discovery tool and reviewed suggested refactorings. Our tool provides similar assistance but focuses on tests, which are simpler to understand than arbitrary code.

The use of static analysis to discover interfaces and constraints (similar in spirit to potential refactorings) in object oriented code has been previously investigated; see representative works by Ammons et al [2] and Whaley et al [43]). Alternatively, automatically-discovered program facts have been leveraged for automatic verification in place of manually written tests. Pradel and Gross use runtime traces to mine for specifications and reporting bugs [28], while Jaygarl et al capture object instances at runtime for mutation testing [16]. Brun and Ernst also explored the possibility of finding errors in code via machine learning in [8] where they found interesting code properties and their violations, which were indicative of developer error.

Kim and Rinard proposed an approach for property verification whereby the developer explicitly defines the property to be tested and verified [21]. This is effective for new development, but does not aid refactoring efforts without significant developer investment, since their approach requires programs to be manually annotated with properties before the properties can be tested.

The closest automated refactoring process for code clones was proposed by Balazinska et al. in [3] Where they automatically refactored code with the java standard libraries based on clone analysis.

6.4 Program Slicing

Program slicing was proposed by Weiser as a method for identifying minimal subsets of a program that could replicate a chosen subset of the program's behaviour [42]. Weiser defined a program slice as "any subset of a program which preserves a specified projection of its behavior."—A slice is a subset of the code in a program, which can reproduce the value contained in a chosen variable at a given location in code. The variable and point in code are called the slicing criteria.

A slice can be of two types to answer two different questions about the code. A backward slice answers the question—"What parts of code are required to correctly execute the current statement?" and a forward slice answers the question—"What parts of code are impacted by the presence of this statement?"

By Weiser's definition, a slice generated by a program slicer should be code which can be run to generate the exact value required for the variable in the slicing criteria. This is useful for generating minimal working programs to replicate a given behaviour however, the resultant code along with side effecting calls is much larger in scope.

Sridharan et al. claim that a smaller slice, containing only lines of code deemed interesting in the given context is adequate for program analysis [35]. This approach to slicing incomplete subsets of the program is referred to as thin slicing.

Slicing is used in testing but not usually in the context of test code analysis. Instead slicing based approaches are popular for limiting data to present to developers. Aggarwal et al. use slicing to aid debugging in [1]. Binkley surveys the use of program slicing in the context of regression testing in [7]. Gallagher and Layman attempted to quantify if cloned slices represent actual clones [14]. They did not find conclusive evidence for or against their hypothesis. We believe that within the context of test code the positive aspects of this approach outweigh the negative aspects. Surendran et al. detect clones in code using a forward slicing approach in code. Their goal was to simplify test design for code which is presently poorly tested [36].

Our approach does not involve slicing when generating assertion protocols; APs are generated for all variables at every program point in code. However, the flow rules of our dataflow analysis limit data collected in a manner that is similar to program slicing. Since APs depend on assignment statements, they are only influenced by code along the use-definition chains of each variable. Hence, the similarity computation is effectively relies on slices along the use-definition chains of the passed parameters.

By selectively comparing APs, the similarity metric is only dependent on a slice of the program used to generate that AP. When we use JUnit assertions and parameter variables to generate our similarity metric, we create a thin, backward slice of the available APs and only consider the APs of the parameters to the assertion rather than the APs of all local variables within the program scope at that point.

As a result, while we do not explicitly use program slicing, the use of APs for clone detection can effectively be transformed into a slicing based clone detection technique.

Chapter 7

Conclusions

In this thesis, we proposed a technique for automatically detecting clones in test code. The approach uses Assertion Protocols containing variable state history. Assertion Protocols are abstractions generated using dataflow analysis which record method calls which influenced a variable's state.

The next step is the computation of a similarity metric between APs to identify likely clones using a graph-based technique to highlighting refactoring opportunities for developers or automated tools.

We use test assertions and the passed parameters as slicing criteria to select appropriate APs to compute a similarity score. Similar test assertions are then grouped and consolidated to report clones at a test method.

Our system provides, as output, a list of test methods (organized into clone sets), along with supporting facts. Developers can use our output to improve code quality in test suites.

We evaluated our approach across 10 open source systems and found that many tests in the systems' test suites are amenable to refactoring. We believe that our work enables the broader use of advanced techniques for writing unit tests, leading to more robust and more concise test suites.

APPENDICES

Appendix A

Data Flow Analysis Definition

Figure A.1 is the formal definition of the data flow analysis as implemented in Soot. It operates on the Jimple intermediate representation where all non-static variables are considered local variables.

The analysis is defined using the following facts:

- DataFlow Direction: The order in which we analysis statements
- Lattice Definition: What facts do we store at each program point
- Initial Estimation: What do we know about the program when we encounter a new artifact
- Entry Point Estimation: What do we know about the program when the analysis begins
- Join Operation: What facts should we preserve when multiple control flows join
- Flow Equations: What facts are added when we process a statement in code

```
Conventions followed:
V × X denotes Cartesian product of V and X
$\mathcal{P}(X)$ denotes the powerset of $X$
$V$ denotes the set of local variables in the method
$AP$ denotes an assertion protocol

Flow Direction: Forward Flow
Lattice: $\mathcal{P}(\mathcal{P}(AP) \times V)$
Lattice elements: $\mathcal{P}(AP) \times V$
Initial Estimation: $\emptyset \times V$
Entry Point Estimation: $\emptyset \times V$
Join: $AP(v) = AP_1(v) \cup AP_2(v)$
Flow Equations:
  if x = y :
    $AP(\texttt{x}) = AP(\texttt{y})$
  if x = y.foo(var1) :
    $AP(\texttt{x}) = AP(\texttt{y}).append(\texttt{resolve(foo)})$
  if x = Constant :
    $AP(\texttt{x}) = \emptyset.append(\texttt{Type(constant)})$
  if y.foo() :
    $AP(\texttt{y}) = AP(\texttt{y}).append(\texttt{resolve(foo)})$
  if x = staticMethFoo(var1) :
    $AP(\texttt{x}) = \emptyset.append(\texttt{resolve(staticMethFoo)})$
```

Figure A.1: This is the formal definition of the data flow analysis as implemented in Soot.

Appendix B

Detailed Test Suite Information

System	**LOC**	**Test LOC**	**Tests at Runtime**	**Tests Methods Analyzed**
Apache POI	85379	57536	1576	4743
Commons Collections	26495	29559	13006	2653
Google-Visualization	8027	9382	784	485
HSQLDB	160665	19136	629	1826
jDom	18916	15428	252	371
jFreeChart	93998	51710	2205	3556
jGrapht	11342	6225	146	297
jMeter	90504	14511	722	982
JodaTime	26881	51497	3688	4746
Weka	246828	13030	3535	1519
MEAN	76904	26801	2654	2118

Table B.1: Code size, test case count (as reported by JUnit) and analyzed test method count.

Appendix C

Output Sample

```
Pattern followed:
Method_Name : Set< Similar_Assertions >
Clone Set:
org.joda.time.chrono.TestBuddhistChronology.void testEquality(): (org.joda.time.chrono.
    TestBuddhistChronology.java:117) (org.joda.time.chrono.TestBuddhistChronology.java
    :118) (org.joda.time.chrono.TestBuddhistChronology.java:116) (org.joda.time.chrono.
    TestBuddhistChronology.java:119) (org.joda.time.chrono.TestBuddhistChronology.java
    :115)
org.joda.time.chrono.TestCopticChronology.void testWithUTC(): (org.joda.time.chrono.
    TestCopticChronology.java:126) (org.joda.time.chrono.TestCopticChronology.java:128)
     (org.joda.time.chrono.TestCopticChronology.java:127) (org.joda.time.chrono.
    TestCopticChronology.java:125)
org.joda.time.chrono.TestEthiopicChronology.void testWithZone(): (org.joda.time.chrono.
    TestEthiopicChronology.java:137) (org.joda.time.chrono.TestEthiopicChronology.java
    :132) (org.joda.time.chrono.TestEthiopicChronology.java:136) (org.joda.time.chrono.
    TestEthiopicChronology.java:134) (org.joda.time.chrono.TestEthiopicChronology.java
    :133) (org.joda.time.chrono.TestEthiopicChronology.java:135)
org.joda.time.chrono.TestGJChronology.void testEquality(): (org.joda.time.chrono.
    TestGJChronology.java:166) (org.joda.time.chrono.TestGJChronology.java:164) (org.
    joda.time.chrono.TestGJChronology.java:163) (org.joda.time.chrono.TestGJChronology.
    java:167) (org.joda.time.chrono.TestGJChronology.java:165)
  ....
  ....
  ....
  ....
```

Figure C.1: Output sample from our tool running on JodaTime.

References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26, May 1991.

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

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 326–336, 1999.

[4] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers*, ESEC-FSE companion '07, pages 513–516, New York, NY, USA, 2007. ACM.

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[6] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[7] David Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(1112):583 – 594, 1998.

[8] Y. Brun and M.D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 480–490, 2004.

[9] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the Workshop on Refactoring Tools*, pages 9:1–9:2, 2008.

[10] Neil Davey, Paul Barson, Simon Field, R Frank, and D Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995.

[11] Ekwa Duala-Ekoko and Martin P Robillard. Tracking code clones in evolving software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 158–167. IEEE, 2007.

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

[13] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[14] K. Gallagher and L. Layman. Are decomposition slices clones? In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 251–256, 2003.

[15] Nils Göde. Evolution of type-1 clones. In *Proc. of the 2009 Ninth IEEE Intl. Working Conference on Source Code Analysis and Manipulation*, SCAM '09, Edmonton, AB, Sept. 2009.

[16] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object capture-based automated testing. In *Proceedings of the international symposium on Software testing and analysis (ISSTA)*, pages 159–170, 2010.

[17] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

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

[19] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[20] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 736 –743, 2001.

[21] D. Kim and M.C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 528–541, 2011.

[22] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, UK, 2001. Springer-Verlag.

[23] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[24] 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*, October 2011.

[25] Seunghak Lee and Iryoung Jeong. Sdd: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 140–141, New York, NY, USA, 2005. ACM.

[26] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[27] E. Merlo, G. Antoniol, M. Di Penta, and V. F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 412–416, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Michael Pradel and Thomas R Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *In Proceedings of the International Conference on Software Engineering (ICSE)*, pages 288–298, 2012.

[29] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: what is that smell? *Empirical Softw. Engg.*, 17(4-5):503–530, August 2012.

[30] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *Transactions on Software Engineering (TSE)*, 30(12):889–903, December 2004.

[31] Chanchal K. Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Per Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

[33] David Saff. Theory-infected: or how I learned to stop worrying and love universal quantification. In *Companion to the SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA)*, pages 846–847, 2007.

[34] David Saff, Marat Boshernitsan, and Michael D. Ernst. Theories in practice: Easy-to-write specifications that catch bugs. Technical Report MIT-CSAIL-TR-2008-002, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, January 14, 2008.

[35] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[36] Anupama Surendran, Philip Samuel, and K Poulose Jacob. Code clones in program test sequence identification. In *Information and Communication Technologies (WICT), 2011 World Congress on*, pages 1050–1055. IEEE, 2011.

[37] Suresh Thummalapenta, Madhuri Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *Proceedings International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 294–309, March-April 2011.

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

[39] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[40] Wei Wang and Michael W Godfrey. We have all of the clones, now what? toward integrating clone analysis into software quality assessment. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 88–89. IEEE, 2012.

[41] Wei Wang and M.W. Godfrey. A study of cloning in the linux scsi drivers. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 95–104, 2011.

[42] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

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

[44] S. Zhang, D. Saff, Y. Bu, and M.D. Ernst. Combined static and dynamic automated test generation. 2011.