2013

Detecting Semantic Method Clones In Java Code Using Method Ioe-behavior

Rochelle Elva
University of Central Florida

STARS Citation

Elva, Rochelle, "Detecting Semantic Method Clones In Java Code Using Method Ioe-behavior" (2013). *Electronic Theses and Dissertations, 2004-2019*. 2620.
https://stars.library.ucf.edu/etd/2620

DETECTING SEMANTIC METHOD CLONES IN JAVA CODE USING METHOD IOE-BEHAVIOR

by

ROCHELLE ELVA
B.S. Computer Science, University of The West Indies, St. Augustine, Trinidad 2002
M.S. Computer Science, University of Central Florida, 2005

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2013

Major Professor: Gary T. Leavens

ABSTRACT

The determination of semantic equivalence is an undecidable problem; however, this dissertation shows that a reasonable approximation can be obtained using a combination of static and dynamic analysis. This study investigates the detection of functional duplicates, referred to as semantic method clones (SMCs), in Java code. My algorithm extends the input-output notion of observable behavior, used in related work [1, 2], to include the effects of the method. The latter property refers to the persistent changes to the heap, brought about by the execution of the method. To differentiate this from the typical input-output behavior used by other researchers, I have coined the term method IOE-Behavior; which means its input-output and effects behavior [3]. Two methods are defined as semantic method clones, if they have identical IOE-Behavior; that is, for the same inputs (actual parameters and initial heap state), they produce the same output (that is result- for non-void methods, and final heap state).

The detection process consists of two static pre-filters used to identify candidate clone sets. This is followed by dynamic tests that actually run the candidate methods, to determine semantic equivalence. The first filter groups the methods by type. The second filter refines the output of the first, grouping methods by their effects. This algorithm is implemented in my tool JSCTracker, used to automate the SMC detection process.

The algorithm and tool are validated using a case study comprising of 12 open source Java projects, from different application domains and ranging in size from 2 KLOC (thousand lines of code) to 300 KLOC. The objectives of the case study are posed as 4 research questions:

1. Can method IOE-Behavior be used in SMC detection?

2. What is the impact of the use of the pre-filters on the efficiency of the algorithm?

3. How does the performance of method IOE-Behavior compare to using only input-output for identifying SMCs?

4. How reliable are the results obtained when method IOE-Behavior is used in SMC detection?

Responses to these questions are obtained by checking each software sample with JSCTracker and analyzing the results.

The number of SMCs detected range from 0–45 with an average execution time of 8.5 seconds. The use of the two pre-filters reduces the number of methods that reach the dynamic test phase, by an average of 34%. The IOE-Behavior approach takes an average of 0.010 seconds per method while the input-output approach takes an average of 0.015 seconds. The former also identifies an average of 32% false positives, while the SMCs identified using input-output, have an average of 92% false positives. In terms of reliability, the IOE-Behavior method produces results with precision values of an average of 68% and recall value of 76% on average.

These reliability values represent an improvement of over 37% (for precision) and 30% (for recall) of the values in related work [4, 5]. Thus, it is my conclusion that IOE-Behavior can be used to detect SMCs in Java code with reasonable reliability.

To my nieces: Philomena and Wanjiku and nephews: Dominic and Muchemi,

never fear to dream, knowing that the only limitation to their fulfillment

is your willingness

to work hard and to persist in spite of obstacles.

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. Gary T. Leavens for his support and guidance in the last three years. I would also like to thank all who have supported and cheered me on in the background. Thanks for believing in me. This is your success too.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

The existence of code duplication (or clones) in software, is a reality, as is evidenced by the number of research papers on software clone detection [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25], and attempts at refactoring code [26, 27, 28, 29, 30, 31, 32] to reduce or eliminate this problem. Software clones have been viewed as problematic because they make software maintenance difficult, introduce bugs in multiple sites in the code [24] and increase the possibility for update anomalies and software aging [33]. This has been of particular concern, because according to existing literature, software maintenance is one of the most—if not the most costly phase of the software life cycle. It is responsible for as much as 80% of the total cost of software [33]. However, in the last couple of years, there has been some debate in the research community on the question of the harmfulness of clones. Originally they were thought by many to indicate "bad smells" in code [34]. Lozano *et al* [35] conducted a study on how clones affected code evolution. While the results were inconclusive, they suggested that the presence of clones in code, negatively impacted maintenance efforts and code evolution. On the other hand, others believe that clones are not really harmful and that maintenance problems thought to be a result of the presence of clones, are instead caused by "sloppy software design practices" [33]. Some researchers even refer to clones as essential [36]. The more commonly held position though, is an acknowledgment that clones need to be monitored and managed and thus detected, but not necessarily re-factored or removed [37].

It is my position that clones are an indication of poor software design and a violation of the good practice of software reuse. They cause code to be sloppy, and bug-prone and add to the cost of maintenance efforts. They should therefore be detected and re-factored to improve the modularity and overall quality of software.

Different approaches have been investigated by researchers both in academia and industry, in an attempt to address the software clone issue and its implications for software maintenance. Some of the strategies developed include the use of string or token-based techniques [33], abstract syntax trees [21] and program fingerprints [17] or metrics [38, 39]. Basit *et al* [40] also present a study of duplication at the structural level, identifying design-level or structural clones.

All of these techniques have had varying degrees of success in accurately detecting syntactic code clones. That is, clones created as a result of copy and paste and therefore look alike or have common text. However, they overlook another more complex clone type—semantic clones. These clones may not be duplicates of code syntax, but represent duplication of functionality. They can be created by code generators, developers who are unaware of methods in existing code and therefore unknowingly re-implement functionality already present in the system. The result is that the code becomes bloated over time, possibly leading to maintenance problems like duplication of efforts in activities such as debugging and analysis.

There has been extensive research on the detection of syntactic clones - that is clones with common text (see Bellon *et al.'s* survey [5]). The applications of this research can be divided into two broad categories: software security and software understanding & maintenance. The former includes research in areas like plagiarism detection [41, 42] and determination of authorship [18], while the latter focuses on clone detection [27] and re-engineering or re-structuring software [26]. However, there is comparatively little research done on the detection and management of functionally similar code—semantic clones; although their existence in real world software is evident in the clone literature. For example, in one study, as many as 405 semantic clones were identified in 589 KLOC of commercial software written in C [4]. In my research 46 semantic method clones (SMC's) are identified in the 1320 methods analyzed in the *Apache* open-source project. Also, in the work done on semantic clones, the reliability measures are not provided for the clones detected or those provided are low.

In addition, most of the tools designed for clone detection are aimed at recognizing only structural similarity of code [5, 10, 24, 33, 43]. This is demonstrated in an experiment using 4 state of the art tools where only 1% of 109 samples of functionally equivalent code are flagged as clones [44]. This is supported by Jiang and Su's work [2] in which they report that in their work on semantic clone detection, 58% of the clones identified were not syntactic clones. Thus in tests such as this, less than half of the semantic clones would go undetected by syntactic based detection tools.

One of the fundamental motivations for the detection and removal of syntactic clones, is their potential for maintenance problems. However, these problems occur due to the semantic similarity of the clones— not just their textual similarity. Thus, the detection of semantically equivalent code is also important. It is my thesis therefore, that analyses for code similarity or duplication, should go beyond syntax checks. The semantics of the code —namely behavior and functionality should also be given consideration. For, if left unmonitored, semantic clones can lead to software quality degradation over time. In addition, this could prove valuable in program restructuring, code reuse and software maintenance. Also, automating the semantic clone detection, would result in a more subjective and reliable process.

The focus of my research is CD for the purpose of software maintenance, which as previously discussed, is responsible for as much as 80% of the total cost of software. The maintenance costs are further increased when code is duplicated (or copied) in multiple places throughout a program's source code, creating what is known as software clones.

1.1 The Problem

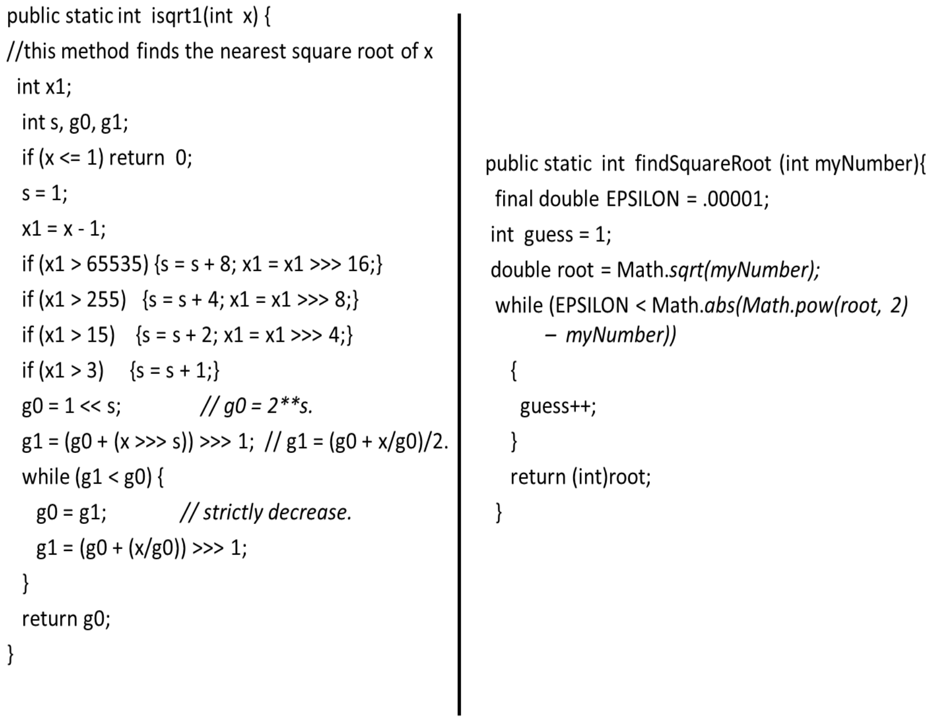
Code duplication in software projects exists in one of two formats: representational or functional. These give rise to syntactic and semantic clones, respectively. Most of the tools designed for clone detection are targeted at syntactic clones. However, the detection of semantic clones is important, since they adversely affect the maintainability of code because of decreased modularity and poor reuse of software. Semantic clones can also lead to degradation in system performance and efficiency. For example, if the functionality implemented by the code is invoked often, given that the computational efficiency of the fragments varies, the net efficiency would be lower than that of the most efficient clone. Hence code would display sub-optimal performance [4].

To my knowledge, in the existing work on semantic clone detection, there are only two tools in the literature [1, 2] for automated detection in source code, one tool which analyzes byte code [45]; another paper that uses manual identification [44] and a tool which applies the concept of functional similarity to identify missed API re-use opportunities [4]. All of these[1] used input-output as the technique for identifying semantic clones. However, this technique is incomplete, since the behavior of a method also includes its effects: changes to the heap including output values and mutations of static and instance fields. The omission of effects allows simpler computations, however, it can imply lower precision and higher incidence of false positives. For this reason, my approach to semantic method clone detection uses information about a method's effect.

[1] Details on the Algorithm used by Keivanloo *et al* [45] were not available

1.2 My Contribution

This study presents an algorithm and tool— JSCTracker, for the automated detection of functionally identical Java methods: semantic method clones (SMCs). In addition to identifying code with similar syntax and similar behavior, it can identify methods such as those in Figure 1.1 which have identical behavior but not the same syntax.



```
public static int  isqrt1(int  x) {
//this method finds the nearest square root of x
 int x1;
  int s, g0, g1;
  if (x <= 1) return  0;
  s = 1;
  x1 = x - 1;
  if (x1 > 65535) {s = s + 8; x1 = x1 >>> 16;}
  if (x1 > 255)   {s = s + 4; x1 = x1 >>> 8;}
  if (x1 > 15)    {s = s + 2; x1 = x1 >>> 4;}
  if (x1 > 3)     {s = s + 1;}
  g0 = 1 << s;              // g0 = 2**s.
  g1 = (g0 + (x >>> s)) >>> 1;  // g1 = (g0 + x/g0)/2.
  while (g1 < g0) {
    g0 = g1;            // strictly decrease.
    g1 = (g0 + (x/g0)) >>> 1;
  }
  return g0;
}
```

```
public static  int  findSquareRoot  (int myNumber){
 final double EPSILON = .00001;
 int  guess = 1;
 double root = Math.sqrt(myNumber);
  while (EPSILON < Math.abs(Math.pow(root,  2)
        − myNumber))
    {
     guess++;
    }
    return (int)root;
 }
```

Figure 1.1: Two Semantic Clones from *Hacker's Delight* [6]

My approach combines the benefits of static and dynamic analysis. The static analysis of a method's type (return type and parameter type list) and effect(persistent changes to the heap), serves as a double pre-filter, to reduce the size of the candidate clone set to be evaluated by potentially expensive dynamic tests. Together, these two types of analyses provide the information on a method's input, output and effects behavior (which I collectively referred to as IOE-Behavior [3]). This information is used to infer semantic equivalence. My tool—JSCTracker is different from any other existing tools, since it employs both static and dynamic analyses in the detection process. I show that the use of static analysis reduces the number of methods to be dynamically tested by an average of 34% compared to methods that do not use the filters [1]. It also identifies SMCs with 68% precision and 76% recall.

1.3 Outline of the rest of the Dissertation

The rest of the chapters are organized as follows:

Chapter 2 gives the definitions for key terms and concepts, required to understand my research. It also offers some background to underlying software concepts that have influenced the design decisions.

Chapter 3 is a detailed description of the algorithm used to detect semantic clones, including the assumptions used in its development. It also includes a description of the tool JSCTracker, used to automate my CD algorithm.

Chapter 4 describes the related work.

Chapter 5 outlines the evaluation of the algorithm and JSCTracker, using a case study of 12 samples of Java open-source software. The evaluation includes the results of the detection process and analysis of the precision and recall of the results.

Chapter 6 presents a discussion of the work covered in this research. It discusses the limitations of the algorithm and presents an analysis of the results. The chapter ends with recommendations for future work.

Chapter 7 gives my conclusions. It summarizes the major findings and the contributions of this work to the research area.

CHAPTER 2
TECHNICAL DEFINITIONS

This chapter defines the primary terms and concepts required to understand this research. More detailed background is provided where necessary, on related work, which has influenced design decisions.

2.1 Clone Detection and Analysis

Clone detection (CD) refers to the process of analyzing source code to identify instances of code duplication also referred to as *software clones* (defined in more detail in Section 2.2 on page 9). Given a sample of source code, the *reference set* or '*reference corpus*' as it is referred to in some literature [5], is the set of all of the clones that actually exist in the source code. This is denoted by R. The set of clones detected by a CD algorithm applied to the source code is referred to as the *returned set* (r). Under ideal conditions, $r = R$. However, in practice, it is possible that r contains false positives or omits false negatives. A *false positive* is a pair of code fragments detected as clones when actually they are not clones. A *false negative* on the other hand is a pair of code fragments which are actually clones, but go undetected. Most CD algorithms, find preliminary candidate clone sets (potential clones). These are further refined to eliminate false positives. The subset of the result set r that are actual clones, that is, are members of R, is denoted by c.

Figure 2.1 shows the relationship between R, r, c, false negatives and false positives.

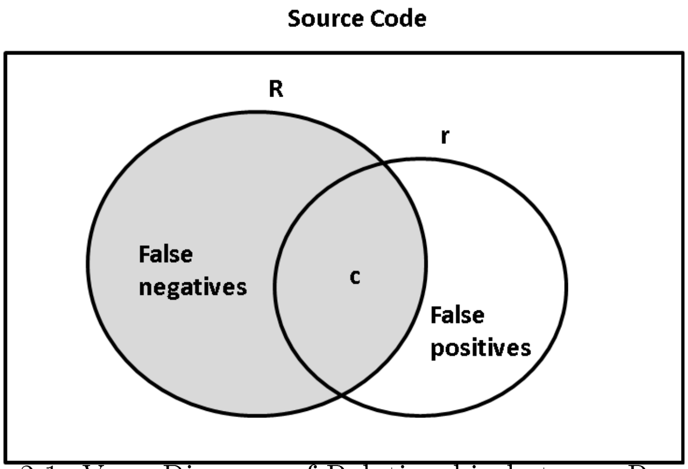


Figure 2.1: Venn Diagram of Relationship between R, c and r

The accuracy or reliability of a CD algorithm is defined in terms of two statistics—precision and recall. **Recall** is the percentage of the reference set of clones that is in the returned set of an algorithm. It is computed using equation 2.1.

$$Recall = \frac{c}{R} * 100 \tag{2.1}$$

Precision is the extent to which a CD technique returns accurate results. It is expressed as the percentage of the returned set that are clones, as shown in equation 2.2.

$$Precision = \frac{c}{r} * 100 \tag{2.2}$$

A perfect CD algorithm therefore has recall and precision values that are 100%.

2.2 Types of Clones

A *Code fragment* is a contiguous bit of code in a sample of source code. It could be a few lines or several lines. Larger fragments are built from multiple atomic units. These include functions, methods, procedures, classes, files and packages. For the purpose of this research, which concentrates on Java, the atomic unit of code fragments is a method. When there is some level of duplication in code fragments, clones result. If the duplication is in the text, then the clones are said to be *syntactic clones.* If the duplication is in functionality or meaning, then the clones are called *semantic clones.* Thus, two code fragments are referred to as *clones* if they satisfy at least one of the following conditions. They are:

- syntactically identical
- syntactically similar
- semantically identical
- semantically similar

Two code fragments that satisfy such a condition are referred to as a *clone pair.* Syntactic clones often result from copy and paste operations. This is not always the case with semantic clones. *Syntactically identical clones* have exactly the same text. While *syntactically similar clones* have text which differs slightly. Semantically identical clones have the same meaning and perform the same function. Thus they can be substituted for each other seamlessly. *Semantically similar clones* perform the same function, most of the time. The differences between these clone types is explained in more detail in the following paragraphs.

Clones are classified into subtypes based on what they have in common. One such taxonomy is that of Bellon *et al.* [5]. They identify 3 classes of clones— *Type I*, *Type II* and *Type III* clones. Baker adds one more class to this list —parameterized clones, as a subclass of *Type III* clones [39, 46]. Later other researchers identified another class of clones—*semantic clones*. Each of these types is described in more detail below.

- *Type I* or *Exact clones*: This is a clone pair which is syntactically identical, meaning that they have the same code text. For example:

Code Fragment 1

```
int Add(int a, int b){
  return (a + b);
}
```

Code Fragment 2

```
int Add(int a, int b){
  return (a + b);
}
```

- *Type II* or *Near miss clones*: This is a clone pair in which the members are only slightly syntactically different, due to less than three minor code modifications, (such as changes in identifier names) after the copy and paste action. In the following example Code Fragments 3 and 4 are near miss clones, since Fragment 3 can be converted to Fragment 4 by a single change (Add → Sum). Code Fragments 3 and 5 are not near miss clones since as many as 5 changes are required to transform Fragment 3 to Fragment 5.

Code Fragment 3

```
int Add(int a, int b){
  return (a + b);
}
```

Code Fragment 4

```
int Sum(int a, int b){
  return (a + b);
}
```

Code Fragment 5

```
int Sum(int x,
     int y){
  return (x + y);
}
```

- *Type III* or *Modified clones*: These clones have similar code so they are syntactically similar, but statements and comments may have been removed or added, or placed in a different sequence. For example:

Code Fragment 6

```
int a = 5;
String str;
int x;
```

Code Fragment 7

```
int x;
int a = 5;
String str;
```

- *Parameterized clones*: This is a subclass of modified clones. They are clone pairs such that there exists a bijective relationship between the identifiers of each of the clones. It is therefore possible to take any member of the pair and by a series of substitutions of the parameters, arrive at a pair of exact clones. For example:

Code Fragment 8

```
int a = 5;
int b = 9;
int x = Change(a,b);
```

Code Fragment 9

```
int x = 5;
int y = 9;
int z = Havefun(x,y);
```

In these two code fragments the parameters *a, b, x* and *Change* in Fragment 8 correspond to *x, y, z* and *Havefun* in Fragment 9. It should also be noted that Code Fragments 3 and 5 in the previous example are also parameterized clones—parameters Sum, x and y can replace Add, a and b to convert code fragment 3 to code fragment 5.

- *Type IV* or *Semantic clones*: Semantic clones are code fragments that are functionally identical. They perform the same function in code and may or may not have any syntactic similarity. Thus, exact clones are often semantic clones; while code fragments which look nothing like each other may also be semantic clones. *Semantic clones* behave the same way and can therefore be used interchangeably in code. For example both code fragments 10 and 11 compute the sum of integers from 1 to 11. Fragment 10 uses a for loop while Fragment 11 uses a while loop, but they both achieve the same end. Hence, they are functionally identical and thus *semantic clones*.

Code Fragment 10

```
int total = 0;
for(int i = 1; i <12; i++){
  total+= i;
}
```

Code Fragment 11

```
boolean notdone = true;
int i = 1;
total = 1;
while(notdone){
  total+= i;
  i++;
  notdone = i < 12;
}
```

2.3 Semantic Method Clones

This research focuses on the detection of semantic method clones (SMCs). Whole methods are selected, since methods are the unit of functionality of object-oriented languages and semantic clones represent functional duplication. Also, detecting behavioral equivalence between arbitrary code fragments is more complex—requiring non-trivial computation of inputs and outputs. In addition, a primary motivation for clone detection is the identification of refactoring opportunities. A functional unit is thus an intuitive choice for refactoring, with least disruption of surrounding code.

The methods in Figure 2.2 are *a semantic method clone pair*. They return double the original value of the field *val* of an object of type *B*, while also updating the field value. The text of the methods is similar, hinting that they may also be syntactic clones. The methods in Figure 2.3 on the following page are also semantic method clones. While their syntax is very different, both of the methods compute the $\log_2$ of an integer parameter.

```
int methodA(B b){
   b.val = b.val*2;
   return b.val;
}
```

```
int methodC(B b){
   int tmp = b.val + b.val;
   b.val = tmp;
   return tmp;
}
```

Figure 2.2: Clones that return twice the value of Object b

```
public static int flp2(int x) {
   x = x | (x >>> 1);
   x = x | (x >>> 2);
   x = x | (x >>> 4);
   x = x | (x >>> 8);
   x = x | (x >>>16);
   return (x - (x >>> 1))
   & 0xffffffff;
}
```

```
public static int HPow2(int x){
  int tmp = x;
  int answer = 1;
  while(tmp > 1){
    answer = 2 * answer;
    tmp = tmp/2;
  }
  return answer;
}
```

Figure 2.3: Semantic clones computing the $\log_2(x)$—adapted from *Hacker's Delight* [6]

The two primary attributes of methods used in this study's detection of semantic method clones are *methodType* and *effects*. Each is discussed in detail in the next two sub-sections.

2.3.1 MethodType of Methods

A method's *methodType*, consists of its return type and the list of types of its formal parameters. Two methods have equivalent methodType if their return type and the sequence of the types of their formal parameters are the same. Table 2.1 on the next page shows six methods and their corresponding methodType information. Methods placed in the same row of the table have an equivalent methodType. For example, in the first row, the methods *checkBalance* and *calcInt* both have a return type of double and an empty parameter list. Thus they have equal methodType. Similarly, the methods of the second row have equivalent methodsTypes since they both return void and take a single parameter of type double. The methods in the third and fourth rows are each the only ones with their methodType, in the given set of methods; since although *getType* and *printState* return the same type— String, they have different parameter lists (empty and Date respectively).

Table 2.1: Sample Methods with MethodType Information

Row	Methods	Return type	Parameter List

1	double checkBalance(){...}
	double calcInt(){...}
		double
		double
			()
			()

2	void withdraw(double amt) {...}
	void deposit (double amt){...}
		void
		void
			(double)
			(double)

3	String getType(){...}	String	()
4	String printState(Date d){...}	String	(Date)

2.3.2 Effects of Methods

A method's *effects* are the set of persistent changes to the heap that result from some execution. This section provides an overview of some existing effect analysis algorithms that influenced the effects analysis used in my semantic method clone detection. The last sub–section, describes the effects analysis used in this research and explains the motivation for the design decisions made. Definitions are also given for important terms and concepts required to understand the effects analysis used.

2.3.2.1 Background

Effect analysis is the determination of the effects of a program, from its source code. This is essentially undecidable, therefore available algorithms are only approximations. However, the goal is to ensure that this approximation is as precise as possible. The effects of a method are determined by a process of static analysis used to track write or update accesses. The result of the analysis is a set of possible effects. Pure methods—methods which do not cause any effects [47] have an empty set of possible effects.[1]

[1] A thorough discussion on the analysis of methods for purity, can be found in the the work of Salcianu and Rinard [47].

Generally, there are two main approaches to effect analysis - *Type-based* and *Refers-to* analysis—commonly called *Points-to* analysis. Both of these approaches follow a basic algorithm: each basic block of the code is visited in turn and the write accesses are collected. The net effect is thus taken as the union of all of these writes. The details of each approach is described in the following paragraphs.

Type-Based Effect Analysis Type-based effects analysis considers all accesses to objects of the same type as accesses to one object. Thus there is no distinction made between different instances of the same class. Razafimahefa [48] discusses two types of Type-based effect analysis— *Class-based* and *Field-based* analysis. Both of these algorithms are conservative, assuming that once an object of a certain type has been read or written to, then the same is true of all objects of that type in the code being analyzed. The distinguishing feature between the two approaches though, is in the level of granularity considered. *Class-based* analysis considers the object as a whole. It does not register reads or writes to individual fields. Thus when a field of an object is written to, the object type is added to the set of effects,[2] independent of the field. In field-based analysis, however, individual object fields are considered. Thus when an object is written to, the object type and the specific field are added to the effects. This latter algorithm requires more memory to store class and field information. However, it provides a more precise approximation than the *class-based analysis*. For example, the code in Table 2.2 on the following page creates two Acct instances and then initializes their data members id, owner and balance. With the *class-based* algorithm, the effects are recorded simply as writes on an Acct (account) object. Lines 3, 4 and 5 contribute $\{Acct\}$to the effects. However, in *field-based analysis*, the effect will be recorded more specifically as writes on the individual fields of an *Acct* object, while ignoring the identities

[2] Razafimahefa also tracks the read accesses in his effects analysis. I do not track the read accesses however, since my research only focuses on persistent changes to the heap. Hence only the variables written to, are significant.

of specific objects. Using *field-based analysis* the following contributions are made to the effects: line 3 adds *Acct.id*; line 4 adds *Acct.owner*; and line 5 adds *Acct.balance*.

Table 2.2: Class-based versus field-based type-based effect analysis of code

Code	**Class-based effects**	**Field-based effects**

`1 Acct a = ...; Acct b = ...;`
`2 .............;`
`3 a.id = ...;`
`4 b.owner = ....;`
`5 a.balance = .... ;`
`6 float x = b.balance;`
	{Acct}	{Acct.id, Acct.owner,Acct.balance}

Refers-to effects Analysis Refers-to analysis recognizes different instances of objects. Thus instead of treating all instances of the same type as one object, the analysis uses a domain that approximately distinguishes individual instances. As with *type-based* analysis, there are two levels of *Refers-to* analyses. The less granular is *aggregate* analysis. In this approach, an object's fields are invisible. Thus writes on a particular field are recorded simply as writes on that particular instance of the object. For example, for the code in Table 2.3, the assignment of values to the *id* and *balance* fields of the *Acct* objects in lines 3 and 5 contribute *a* to the effects, while the assignment in line 4 adds *b*. The field-based version of the Refers-to effects analysis is called the *with-field* analysis. In this analysis both object instances and individual fields are considered when the write effects are recorded. For example, in Table 2.3, lines 3 and 5 contribute *a.id* and *a.balance* to the effects; while line 4 contributes *b.owner*.

Table 2.3: Aggregate versus with-field Refers-to effect analysis of code

Code	**Aggregate effects**	**With-Field effects**

`1 Acct a = ...; Acct b = ...;`
`2 .............;`
`3 a.id = ...;`
`4 b.owner = ....;`
`5 a.balance = .... ;`
`6 float x = b.balance;`
	{a,b}	{a.id,b.owner, a.balance}

The advantage of this type of effects analysis is that it is more precise than type-based analysis, however, it requires more memory storage and processing time.

Effects of Method Calls In both of the approaches to effects analysis (Type-based and Refers-to) just described, if the statement or expression that constitutes a basic code block contains a method call, then the effects of that call will be the union of the effects of the called method and the effects of any other sub-expressions found in the block. An example of this computation is shown in Table 2.4.

Table 2.4: Computation of effects for method including a method call

Method	**Effects**

`public void MergeBalance(Acct a,Acct b)`
`{`
`   b.balance = b.balance + a.balance;`
`}`
	{b}

`public void UseMergeBalance(Acct a,Acct b, Acct c)`
`{`
`  c.balance = MergeBalance(a,b);`
`}`
	{b, c}

In the *MergeBalance* method (in Table 2.4), the set of objects written to only has one member *b*, since this object is mutated in the assignment statement. Thus, the net effect of method *MergeBalance* is given by equation 2.3.

$$MergeBalance.getEffects() = \{b\} \tag{2.3}$$

The method *UseMergeBalance* calls *MergeBalance*, thus, the effects of the former include those of the latter. In addition, UseMergeBalance also writes to object *c* in the assignment statement. Thus the effects of *UseMergeBalance*, are the effects of *MergeBalance* $\bigcup \{c\}$.

Hence, the net effects of *UseMergebalance* are given by equation 2.4.

$$UseMergeBalance.getEffects() = \{b, c\} \tag{2.4}$$

2.3.2.2 My Program Analysis for Method Effects

Method effects analysis is used in this research as a preliminary filter, to quickly and efficiently generate a set of candidate (potential) clones, which will be more accurately refined using dynamic testing. For this reason, the field-based version of Type-based analysis (described in Section 2.3.2.1 on page 15) is used in my research. The Type-based approach is selected over the Refers-to analysis, since it is a quicker algorithm, requires less memory, yet provides adequate information to differentiate between methods. The Type-based analysis will not exclude any candidates that would identified by the Refers-to analysis. Instead, it provides a greater over-approximation which suffices at this stage in the algorithm; since dynamic testing is available as the final filter to remove imprecision due to over-approximations. In this context, speed and efficiency are valued over detail and precision.

The field-based, Type-based, effects analysis used in this research has been modified to analyze whole methods and record only mutations, while ignoring read accesses. Only object parameters and mutated non-local variables are tracked in this analysis. This decision is made since only these mutations can alter the heap in a way that persists beyond the method. Thus unnecessary computations are avoided.

For this research a static effects analysis called *non-local variable mutation* analysis (NLVM) is developed based on Razafimahefa's work [48]. When analyzing the effects of a method, for each program point (or line of code), this analysis tracks all non-local variables, including object fields, that have been mutated up to that point.

The equations for NLVM$_{entry}(l)$ and NLVM$_{exit}(l)$ represent the set of non-local variables mutated before or after the execution of the l^{th} program point (think line of code) respectively. The domain of the NLVM analysis is the power set of the set of pairs of labels in the code, L, and variables (non-local variables, receiver object fields and object parameters mutated in the method) V, as shown in equation 2.5.Any *NLVM* equation therefore returns a set -of pairs of a label and a variable (mutated non-local variable). The label refers to the line of code where that variable was last mutated, up to the current point in the code. Thus intuitively, for all program points l, NLVM$_{entry}(l)$ is a subset of the exit value for the previous program point, as shown in equation 2.6. The $NVLM_{entry}$ (l) refers to effects that are possible up to the start of the l^{th} program point. New variable—label pairs are then added (of the form variable identifier, l) for all variables mutated in l. Thus in general, $NVLM_{exit}$ (l) is the set of definitions that existed at the entry into l, union any variable mutated in l.

$$\mathcal{P}(L \times V) \tag{2.5}$$

$$NLVM_{entry}(l) \subseteq NLVM_{exit}(l\text{-}1) \tag{2.6}$$

For example, the code for a method to scale a Square object is shown in Figure 2.4 on the next page. The equations for the NLVM analysis for that code are given in Figure 2.5 on the following page. The parameter list for this method contains no objects. Thus the NLVM analysis will only track non-local variables in the body of the method. There is only one—the receiver object which is of type Square. Since this is a *type-based, field-based analysis*(see Section 2.3.2.1 on page 15), object variables will be tracked as the *classname.fieldname.*

Static variables will be tracked as *classname.variablename*; where *classname* is the type of the static variable.

```
1 public void scale(int x){
2         if(x > 2){
3                 int tmp = width * x;
4                 this.width = tmp;
5                 area = width * width;
6         }
7 }
```

Figure 2.4: Sample method for effects analysis

NLVM$_{entry}$(2) = {}
NLVM$_{entry}$(3) = NLVM$_{exit}$(2)
NLVM$_{entry}$(4) = NLVM$_{exit}$(3)
NLVM$_{entry}$(5) = NLVM$_{exit}$(4)

NLVM$_{exit}$(2) = NLVM$_{entry}$(2)
NLVM$_{exit}$(3) = NLVM$_{entry}$(3)
NLVM$_{exit}$(4) = NLVM$_{entry}$(4) $\bigcup$ {(Square.width, 4)}
NLVM$_{exit}$(5) = NLVM$_{entry}$(5) $\bigcup$ {(Square.area, 5)}

Figure 2.5: NLVM equations for program in Figure 2.4.

Thus for the code in Figure 2.4, NLVM$_{entry}$(2) = {} since no code has been executed yet and there are no object parameters. NLVM$_{exit}$(2) is also equal to {}, since there is no mutation in line 2. Likewise, for line 3, NLVM$_{entry}$(3) is the empty set since it is equal to NLVM$_{exit}$(2). Also, since there are no non-local mutations in line 3, the exit value is also empty. Thus NLVM$_{exit}$(3) is also equal to {}. New variable—label pairs are then added of the form (variable identifier,l) for all variables mutated in l. For example, for label 4 in the code in Figure 2.4, NLVM$_{exit}$(4) is computed as shown in equation 2.7.

$$\begin{aligned} NLVM_{exit}(4) &= NLVM_{entry}(4) \bigcup \{(Square.width, 4)\} \\ &= \{\} \bigcup \{(Square.width, 4)\} \\ &= \{(Square.width, 4)\} \end{aligned} \tag{2.7}$$

The full effects analysis for the method *scale* shown in Figure 2.4 on the preceding page, is given in Table 2.5.

Table 2.5: Effects of the method *scale* of Square class in Figure 2.4 on the previous page

l	NLVM$_{entry}$*(l)*	NLVM$_{exit}$*(l)*
1	{}	{}
2	{}	{}
3	{}	{}
4	{}	{(Square.width,4)}
5	{(Square.width,4)}	{(Square.width,4),(Square.area,5)}

2.4 Notions of Equivalence

This section defines the notions of equivalence used in the detection of semantic method clones (SMCs). It explains what it means for two methods to be semantically the same and how this is measured.

2.4.1 Semantic Equivalence

The identification of SMCs is essentially the problem of detecting *semantic equivalence* between methods. A pair of methods with the same observable IOE-Behavior for all input values, is said to be *semantically equivalent.* A *method's IOE-Behavior* is its input, output and effects behavior. Input refers to state of the heap and the value of parameters when the method is called. Output refers to the value returned by the method if it is non-void or nothing for void methods. A method's effects are the changes that its execution causes to the heap and which persist after the method call. Effects are discussed in detail in Section 2.3.2.2 on page 18. A method's effects can include updates to fields of its receiver object (for non-static methods) and/or updates to static fields and mutations of object parameters. *Semantic equivalence* between two methods can thus be summarized as:

1. ∀A ∀B | A,B ∈ Methods

   $$\textit{IOE-Behavior}(A) = \textit{IOE-Behavior}(B) \Longleftrightarrow \textit{SemanticEquivalent}(A, B) \tag{2.8}$$

   Two terms A and B are considered to be *contextually equivalent* if they are both valid inputs for any given context C[ ] and in that context they return the same value. Thus C[A] = C[B]. This means that they can be transparently substituted for each other in a larger program. Hence another way of defining semantic equivalence in clones, is to say that

2. Two code fragments are *semantically equivalent* if they are *contextually equivalent*.

2.4.1.1 Defining Semantic equivalence in Classic Java

The concept of semantic equivalence of methods is demonstrated in more detail using Classic Java [7]. This is a small language that contains the basic elements of Java. Due to its relatively small size compared to Java, it can be used to demonstrate concepts involved in the determination of semantic equivalence generally and concisely. The syntax for Classic Java is given in Figure 2.6 on the next page.

In order to measure semantic equivalence between methods, the two properties *method-Type* and *effects* need to be computed for the method candidates. The rules for computing these attributes for the different Classic Java AST nodes are given in Figure 2.7 on page 24 and Figure 2.8 on page 25 respectively.

$\texttt{P}$ = $defn^{*}e$

$\texttt{defn}$ = class c extends c implements i^{*} $\{field^{*}\ meth^{*}\}|$ interface i extends i^{*} $\{meth^{*}\}$

$\texttt{field}$ = $t\,fd$

$\texttt{meth}$ = $t\ md(arg^{*})$\{body\}

$\texttt{arg}$ = $t\ var$

$\texttt{body}$ = e | abstract

$\texttt{e}$ = new c | var | $null$ | $\underline{e : c}.fd$ | $e.md(e^{*})$ | super $\underline{\equiv \text{this: } c}.md(e^{*})$ |
view $t\ e$ | let $var = e$ in e

$\texttt{var}$ = a variable name or *this*

$\texttt{c}$ = a class name or Object

$\texttt{i}$ = interface name or Empty

$\texttt{fd}$ = a field name

$\texttt{md}$ = a method name

$\texttt{t}$ = c | i

Figure 2.6: Syntax of Classic Java quoted from Flatt et al. [7]

In Figure 2.7 the methodType attribute is defined for all of the nodes except for the field, arg and body nodes, since as shown in Figure 2.6 these do not contain a method declaration. The methodType equations all return a set of methodTypes declared in the particular AST node. Each methodType contains return type and parameter type information as defined in Section 2.3.1 on page 13. For a method, this equation simply returns its methodType as a set. The methodTypes for a *defn* (class or interface), are computed as the union of all of the methodTypes of its declared methods.

```
P.getMethodType() →
          switch (P){
            case e:
              e.getMethodType()
            case (defn$_1$...defn$_n$)e:
               $\bigcup_{i=1}^{n} defn_i$.getMethodType() $\bigcup$ e.getMethodType
            default:
              empty set}

defn.getMethodType() →
          switch (defn){
          case  class c$_i$ extends c$_j$ implements $i^*$ {field$^*$ meth$_k^*$}:
             $\bigcup_{k=0}^{n}$ meth$_k$.getMethodType()
          case i extends i$^*$ {meth$_k^*$}:
             $\bigcup_{k=0}^{n}$ meth$_k$.getMethodType()
          default:
            empty set}

meth.getMethodType() →
        switch(meth){
        case t md {body}:
           new methodType(t)
        case t md arg$_1$...arg$_n${body}:
           new methodType(t,arg$_1$...arg$_n$)
        default:
           empty set}

e.getMethodType() → {new MethodType()}
```

Figure 2.7: Part A: Rules for computing methodType for Classic Java AST nodes

P.*getEffects*() →

```
switch (P){
  case e:
    e.getEffects()
  case (defn$_1$...defn$_n$)e:
    $\bigcup_{i=1}^{n} defn_i$.getEffects() $\bigcup$ e.getEffects()
  default:
    empty set}
```

defn.*getEffects*() →

```
switch (defn){
 case  class c$_i$ extends c$_j$ implements $i^*$ {field$^*$ meth$_k^*$}:
    c$_j$.getEffects() $\bigcup_{k=0}^{n} meth_k$.getEffects()
 default:
   $\top$ }
```

meth.*getEffects*() →

```
switch(meth){
case t md arg$^*${body}:
   body.getEffects()
default:
    $\top$ }
```

field.*getEffects*() → `empty set`

arg.*getEffects*() → `empty set`

e.*getEffects*() →

```
switch(e){
case new c:
   (constructor of c).getEffects()
case  e$_i$.md(e$_j^*$):
   e$_i$.getEffects()$\bigcup$ md.getDef(md,typeof($c_i$)).getEffects() $\bigcup_{j=0}^{n} e_j$.
       effects()
case  c.md(e$_j^*$):
   md.getEffects() $\bigcup_{j=0}^{n} e_j$.effects()
default:
     empty set}
```

body.*getEffects*() →

```
if(isAbstract)
    $\top$
else
   e.getEffects()
```

Figure 2.8: Part B: Rules for computing effects for Classic Java AST nodes

The effects equations for Classic Java are given for all of the AST nodes. This is shown in Figure 2.8 on the preceding page. For each type of node, the effect information contains pairs of the object type and fields that it can mutate—that is both instance and static fields. Examples of the attribute definitions and equations used in this research are shown in Section 3.3 on page 41. Since the methods of an interface are all abstract, the effects of the methods of an interface are given as a set called $\top$ (top) each of whose elements is unknown. This is an over-approximation of the effects of the interface so that candidates would not be filtered out in the filtering stage, although they may be when dynamically tested. The same value $\top$ is used for abstract classes and methods. For other classes, the effects are computed as the union of the effects of the super class (if any), and the collective effects of the class methods.

For a method, the *getEffects* equation returns the method's effects computed as described in Section 3.4.2 on page 49. The effects of an abstract method have the value $\top$. For an argument and a field, this returns an empty set, since *arg* and *fields* in Classic Java do not contain method calls.

For an expression(e), the *getEffects* method returns an empty set, except for the method call expressions (including calls to constructors using new). When the method call is made with a dot expression and the left side is a class name, the effects are computed as the union of the effects of the called method and arguments. When the left side of the dot expression is another expression, then the effects are computed as the union of the effect of the expression, called method, and arguments. The effect of a constructor call is evaluated as the effects of that constructor method. The effects of a call to a super method, are evaluated as the effects of the method in the super class. The effects of a *body* node are returned as the effects of its contained expression or $\top$ if it is abstract. Figure 2.9 on the next page gives the equations for creating the program MethodTable. The MethodTable equations return a lookup table for the methods in a *defn* or program. The lookup table contains an entry for each method, including its class information, location, size, type (static or non-static) and methodType and effect attribute values. For a program, this stores entries for all of the methods declared in that defn. It is created from the union of the MethodTables of all of the class *defn's* that make up the program. The interface *defn* has a MethodTable in which all methods have effects values of ⊤, since all of the methods are abstract. Methods have a MethodTable with only one entry and that is the information for that specific method.

```
P.getMethodTable() →
          switch (P){
            case e:
              e.getMethodTable()
            case (defn$_1$...defn$_n$)e:
               $\bigcup_{i=1}^{n} defn_i$.getMethodTable() $\bigcup$ e.getMethodTable()
            default:
              empty set}

defn.getMethodTable() →
          switch (defn){
            case  class c$_i$ extends c$_j$ implements $i^*$ {field$^*$ meth$_k^*$}:
             c$_j$.getMethodTable() $\bigcup_{k=0}^{n}$ meth$_k$.getMethodTable()
            default:
              empty set}

method.getMethodTable() →
        new HashMap<TypeEffect,methodEntry>().add
          (MethodTypeEffect(this.getMethodType(),this.getEffects()),
           new MethodEntry(this.getClass(),this.getID,this.isStatic(),
           this.start(), this.end()))
```

Figure 2.9: Part C: Rules for computing MethodTable for Classic Java AST nodes

2.4.2 Structural Equivalence

When checking for semantic equivalence, at times it is necessary to be able to determine if the objects returned by two methods are equal; or if receiver objects of methods and/or parameter objects or static field objects have been altered in the same way by method executions. The use of the Java *Object.equals(Object)* method is not always appropriate in these instances, since, unless overridden, this method determines equivalence by checking if two objects point to the same location in memory. However, for the purposes of this research, the desired notion of object equivalence, is a *structural equivalence* check. That is a check that corresponding object fields or data member values (and not necessarily memory locations), are equal. Generally in Java, there are three ways in which we can think about equivalence between two objects: identity, observational and structural equivalence.

Identity means that two objects have the same memory address. However, if two methods return the same value, which is an object (meaning that the value of corresponding fields are the same), but the two objects are stored in different memory locations, identity will not recognize the two results as equivalent. For example during the dynamic testing phase for an equivalence class, my algorithm creates new objects to test each method. The objects created are structurally equivalent so that methods are called on objects in the same state. However, these objects cannot be expected to have the same identity. Consequently, if identity is used as the measure of equivalence, false negatives may result. Hence, identity is not an appropriate measure of equivalence for this research.

Observational equivalence between two methods means that a program cannot be written to differentiate between them. While this notion of equivalence may return correct results for my algorithm it is unsuitable, since observational equality cannot be computed easily, or efficiently. This leaves only one other option for evaluating object equivalence—Structural equivalence.

Two objects of the same class are defined as *structurally equivalent*, if each of their corresponding fields stores a structurally equivalent value; the base case is that primitive values are structurally equal if they are equal values. This means that their values are the same, even if their locations in memory are different. Corresponding object fields of structurally equivalent objects are also structurally equivalent. This is precisely the notion of equivalence that is required for comparing objects in my CD algorithm. For this reason, the notion of structural equivalence is used in this research.

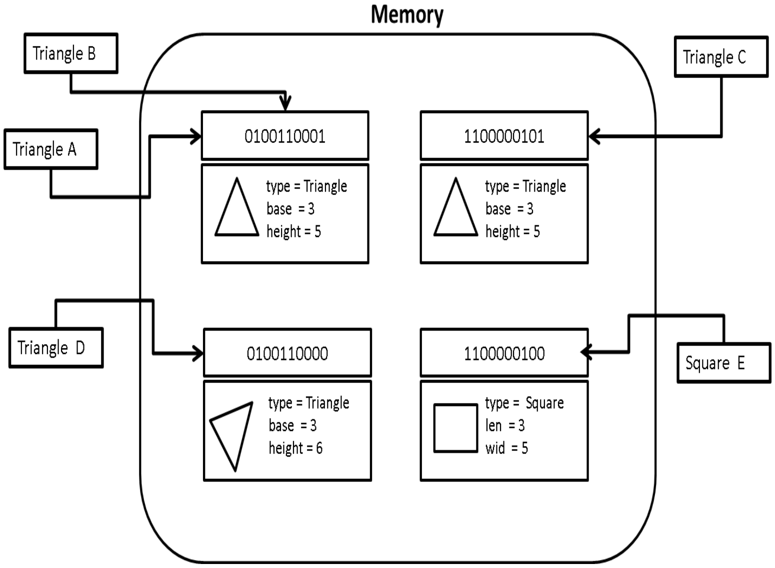


Figure 2.10: Example showing structural equivalence between objects

For example, in Figure 2.10 *Triangle A* and *Triangle B* would return true using the Java *Object.equals* method, since they point to the same address. They are also structurally equivalent, since they point to the same object and an object is equal to itself. On the other hand, the Java *Object.equals* method would return false for *Triangle B* and *Triangle C* since they reference different memory locations. However, these two Triangles are structurally equivalent, given that the corresponding type, base and height field values are all equal. In addition, *Triangles A, B* and *C* are also structurally equivalent. *Triangle D* is not structurally equivalent to the others, since its height value is different. *Square E* is also not structurally equivalent to any of the other objects, as it belongs to a different class and its fields are all different.

2.5 JastAdd

JastAdd is a Java-based compiler generator developed by Hedin *et al.* [8]. It allows developers to extend Java-like languages by adding attribute descriptions and equations to compute them. These new descriptions add attributes based on the language's AST—for my case, the Java AST (a sample of which is shown in Figure 2.11 on the next page).

Each AST node is a class. New attribute descriptions are described in static JastAdd aspect files similar in syntax to AspectJ aspects. Basically two types of attributes can be described: synthesized and inherited. The *synthesized attributes* are defined for each node and are based on the content of the node. *Inherited attributes* are defined in the parent node and propagated down the AST to its children nodes. The computation of both types of attributes is defined using equations which are like Java methods. *Synthesized attribute* equations are like abstract virtual methods. The attributes themselves are like Java instance fields.

Each node in the AST is of the form shown in equation 2.9.

$$ASTnodename : type ::= components \tag{2.9}$$

```
1 abstract Stmt;
2 abstract BranchTargetStmt : Stmt;  // a statement that can be
     reached by break or continue
3 Block : Stmt ::= Stmt*;
4 EmptyStmt : Stmt;
5 LabeledStmt : BranchTargetStmt ::= <Label:String> Stmt;
6 ExprStmt : Stmt ::= Expr;
7 SwitchStmt : BranchTargetStmt ::= Expr Block;
8 abstract Case : Stmt;
9 ConstCase : Case ::= Value:Expr;
10 DefaultCase : Case;
11 IfStmt : Stmt ::= Condition:Expr Then:Stmt [Else:Stmt];
12 WhileStmt : BranchTargetStmt ::= Condition:Expr Stmt;
13 DoStmt : BranchTargetStmt ::= Stmt Condition:Expr;
14 ForStmt : BranchTargetStmt ::= InitStmt:Stmt* [Condition:Expr]
     UpdateStmt:Stmt* Stmt;
15 BreakStmt : Stmt ::= <Label:String>;
16 ContinueStmt : Stmt ::= <Label:String>;
17 ReturnStmt : Stmt ::= [Result:Expr];
18 ThrowStmt : Stmt ::= Expr;
19 SynchronizedStmt : Stmt ::= Expr Block;
20 TryStmt : Stmt ::= Block CatchClause* [Finally:Block];
21 AssertStmt : Stmt ::= first:Expr [Expr];
22 LocalClassDeclStmt : Stmt ::= ClassDecl;
23 VarDeclStmt : Stmt ::= Modifiers TypeAccess:Access VariableDecl
     *;
24 VariableDeclaration : Stmt ::= Modifiers TypeAccess:Access <ID:
     String> [Init:Expr]; // Simplified VarDeclStmt
25 abstract CatchClause ::= Block;
```

Figure 2.11: Java AST statement nodes quoted from Hedin *et al.* [8]

Where *nodename* is the type of node. For example in Figure 2.11 line 20, the AST node is the '*TryStmt*'. Its type is *Stmt*, indicating that this node is a Java statement. The components are other Java AST nodes which are used in the *TryStmt*. Components can be given special names other than the Java AST node type that they belong to. When such a name is used, the component has the format shown in equation 2.10.

$$specialname : Java\ \ AST\ \ node\ \ type \tag{2.10}$$

In the *TryStmt* definition in Figure 2.11 line 20, the AST node components are: *Block* defined in line 3, *CatchClause* of which there may be 0 or more (defined in line 25); and another optional *Block* differentiated from the first with the special name '*Finally*' (The square braces indicate that the component is optional). Each of these components is an attribute of the node and can be accessed using accessor functions. The naming convention for the accessor functions use the word *'get'* followed by the name of the component type, unless it has a special name, then get is followed by the special name. In the *TryStmt* example, the first Block is accessed using the method: *getBlock()*. The second block is accessed using the method *getFinally()*. Since this is an optional component, JastAdd provides the boolean method to check its existence—*hasFinally()*. Since there can be multiple CatchClauses the method *getNumCatchClause*() is available for finding how many are contained in the *TryStmt*. The list of CatchClauses is returned by the method *getCatchClause()*.

2.6 Random Sampling

In dynamic testing, the entire population of all possible inputs (including heap states) of a method is not available. Thus the selection of a representative sample is important. For dynamic testing, the *population* is the domain of all possible combinations of input values and system states of the methods under investigation. In such testing, a good *sample* is a subset of the population, that is representative of the entire set. Such a set is difficult to create. However, good research practice strives at ensuring that the differences between the population and sample are minimized as much as possible, and that they are the result of chance, instead of flaws in the selection process [49]. This is one of the challenges of the research in this dissertation, since it was not possible to test the entire domain of inputs for the methods being investigated. To deal with this challenge random sampling is used for selecting a sample.

Random sampling is considered to be the best way of selecting a sample guaranteed to be representational of the entire group [49]. In this strategy, a subset is chosen from the whole population such that each element has an equally likely chance of being chosen. Thus the sampling is fair. The primary advantage of this selection process is that there is a high probability that the selected sample is truly representative of the whole population. This probability, however, is directly proportional to the size of the selection. Thus, the larger the sample, the more likely it is to be representative of the *population.* One down side is that special elements of the group which add an interesting dimension to the characteristics of the whole, may not be selected in the sample. Consequently, some important aspects may go untested.

An alternative sampling strategy which addresses guaranteeing that special elements of the population are included, is *stratified random sampling.* This strategy begins by identifying subgroups within the whole population and then selecting a sample, by randomly selecting from each sub-group. To ensure that the resulting sample is truly representative of the entire group, the number of elements selected from each subgroup, is the same proportion as exists in the whole population.

For example, if the performance of 100, male and female, 5th graders in a Math exam is under study. A random sample of 10 of the students would not necessarily include both groups (male and female). To guarantee that both are included, the sub-groups of boys and girls are created and students chosen from both groups randomly. If for example there are 40 boys and 60 girls. To collect a stratified sample of 10 students, 4 boys and 6 girls need to be selected, so that the proportions in the sample are representative of the whole.

For my research I use a modification of stratified random sampling. This strategy allowed me to consider special input value cases (shown in Table 2.6 on the following page) such as 0 and null (which could produce atypical method behavior and possibly distinguish between borderline clones), and other regular values. Large samples of the input are also generated for testing, to increase the probability that the sample is representative of the domain. Since the input domain can be infinite, the size of the whole population is also infinite. Thus, it is not possible to strictly maintain the relative proportions of special and regular cases, in the sample. Instead, since the average number of special cases is approximately 4, I select 2 special cases for *numTest* (number of tests run) values 1–30 and 3 for more than 30 tests (except when the number of special cases for a particular type is less than 2. Then all are selected. For example there is only one special value for objects—null. So this value is always evaluated). This was to ensure that the special cases were sampled since there was the probability that they might differentiate between borderline clones.

Table 2.6: Special values for Java types

Type	Special value cases
byte	Byte.MIN_VALUE, Byte.MAX_VALUE
double	Double.NaN, Double.NEGATIVE_INFINITY, Double.MIN_VALUE,0, Double.MAX_VALUE, Double.POSITIVE_INFINITY
float	Float.NaN, Float.NEGATIVE_INFINITY, Float.MIN_VALUE,0, Float.MAX_VALUE, Float.POSITIVE_INFINITY
int	Integer.MIN_VALUE, -1, 0, 1, Integer.MAX_VALUE
short	Short.MIN_VALUE,0, Short.MAX_VALUE
String	null, “"
long	0,Long.MIN_VALUE, Long.MAX_VALUE
object	null

2.7 Privileged Aspects

In this research, semantic equivalence between methods is ultimately determined by dynamic tests. This involves calling the candidate clone methods within the test file and evaluating the output—result and effects. This presents a particular challenge for private and protected methods which according to Java visibility rules, cannot be called outside of their declaring class or its package. Since these methods also have to be analyzed for semantic similarity, there must be the facility for calling them in the test file. Aspects—particularly, *privileged aspects* provide this access.

Aspects are class-like files, used to provide the implementation of some concern (property or behavior), which spans different object types. Like classes, they can encapsulate methods, and field declarations. Standard aspects follow variable and method scoping and visibility rules similar to Java classes; and normally may not be able to access methods defined as private. In some situations, such as when analyzing for semantic clones, it may be necessary to relax these access rules to analyze private methods for semantic equivalence. A special subclass of aspects called *privileged aspects* have this capability. Using the keyword **privileged** in the aspect declaration, grants it access to private methods and data members[3]. Another major advantage of aspects, is that they provide the means to change object behavior by modifying a class outside of that class declaration. Aspects can be used to declare new methods and even fields for a class. For example, the *runTest* method (described in detail in Section 3.5.3 on page 63) for each equivalence class of candidate clones, is declared in an aspect. This method is defined outside of the equivalence class, in a privileged aspect, since it contains method calls to all methods of the equivalence class, and some of these methods may be declared private in their declaring class.

The syntax of an aspect shares some similarity with a Java class. However, they also include aspect-specific components such as *pointcut*, *advice*, and *inter-type* declarations. These features are used to facilitate access to specific points in the code and to define the

[3] A more in depth study of *privileged* aspects with examples can be found in [50]

actions to be taken when such points are reached. Each of these features is described in the following paragraphs.

Join points are identifiable points in a program such as the initialization of a data member or a method call. *Pointcuts* are the predicates defined in an aspect to match specific sets of join points. So pointcuts are written to pick out specific join points like method calls, or object field accesses and to identify values or states of the system at that point. An *advice* is associated with some pointcuts. It describes the action to be taken— that is the code that has to be executed when a particular pointcut is encountered. The type of action to be taken is determined by advice keywords. For example, at a join point, *before* and *after* advice have to be executed before and after the code of the join point respectively. An *around* advice defines the action that has to be taken instead of executing the code of the join point.

For example, a banking system might want to track how much money it pays out to customers for the day. The system has some accounts (Acct objects), owned by its customers. The methods *deposit* and *withdraw* are available for all *Acct* objects. A *pointcut*:

$$pointcut\ \ call(acct.withdraw(double))$$

is defined for the system. This pointcut will identify all points in the code where there is a call to *withdraw* from an *Acct* and a *double* parameter is passed. This pointcut can be named as in the following code. It is the same pointcut as previously described but it has been given a name—*withdrawal.*

```
pointcut withdrawal(Acct a, double amt): call(void Acct.withdraw(
    double)) && target(a) & args(amt)

after(Acct a, double amt): withdrawal(a, amt) {
  cashOut+= amt;
}
```

It takes two parameters the Acct receiver object and the amount to be withdrawn passed as an argument to the *withdraw* method. An advice is also defined for the pointcut *withdrawal.* This advice updates the system variable *cashOut* after the execution of the join point which matches the point cut. For example considering 3 customers with accounts *a-c* and they each withdraw $10, $20, $30 respectively from their accounts. This would be reflected by the following code:

```
a.withdraw(10);
b.withdraw(20);
c.withdraw(30);
```

The pointcut would identify these three withdrawals and invoke the advice after each execution of the call to *withdraw.* This would result in the following execution:

```
a.withdraw(10);
cashOut+=10;
b.withdraw(20);
cashOut+=20;
b.withdraw(30);
cashOut+=30;
```

2.8 Instrumentation in Java

When the type of a formal parameter is an abstract class or an interface, instantiation presents a problem, since neither kind of type can be instantiated. In such cases, a non-abstract subclass or an implementing class needs to be used to create the test instance. This can be achieved by querying the Java virtual machine (JVM) at any given time during runtime, to check the set of loaded classes to find a suitable type from that list. This functionality is provided by the Java Instrumentation interface introduced into the JDK (Java Development Kit) 1.5 and later. Instrumentation allows analysis tools like JSCTracker to be able to passively access the state of the JVM. This query is discussed in more detail in Section 3.5.1 on page 53.

A major and final step in the semantic clone detection process described in this research is testing. This incorporates simulating contexts and generating parameters as described in detail in Section 3.5.1. Using these simulated values, method calls are constructed for the evaluation of the methods under test. Instrumentation is used in this research, to identify usable sub-types for parameters with types that are abstract classes or Interfaces. The process of querying the JVM is costly thus steps are taken to minimize the overhead. These are discussed in detail in Section 3.5.1.

The instrumentation instances can be generated in either of two ways. In one case, the instance is created prior to the launch of the JVM. This is achieved by defining a Java *agent* class with a method *premain*. When the JVM attempts to launch, it detects the presence of the agent class, and recognizing the premain method, its regular activities are interrupted and instead it creates an instance of the instrumentation class, loads it and returns this instance to the premain method, via the instrumentation parameter. This approach is useful when the monitoring activities are required prior to the execution of the application's *main*. However, in instances when the monitoring is required to occur as part of the execution of main, (as for the purposes of this current research), the instrumentation instance is created after the JVM has been loaded. In such a case the agent class must provide an *agentmain* method. As before, the JVM creates an instance of the Instrumentation class, loads it and returns that value to the agent method. Once this instrumentation has been made available to the agent, it can now be accessed and its interface member-methods invoked as with any object. This can be done within the agent or even within the Java application, by making a static method call or accessing a static variable. A list of all of the methods of the Instrumentation interface can be found in the javadoc available in an online document provided by Oracle [51].

CHAPTER 3
ALGORITHM

This chapter begins with a explanation of the underlying assumptions of this research and the rationale for them. It continues with a detailed description of the stages of my method IOE-Behavior algorithm for semantic method clone (SMC) detection.

3.1 Assumptions

In the course of implementing the SMC detection algorithm used in this research, a few assumptions were made because, although not supported by a formal proof, they were intuitive or narrowed a scope that might have been too wide to investigate reasonably. These assumptions are discussed below. The impact of these assumptions on the precision of my experimental results is discussed in Section 6.1 on page 98.

The first assumption is that the possible persistent changes brought about by a method's execution are limited to: changes to receiver objects, class variables and parameter objects. Virtual machine states and effects on external devices such as the system clock, are not considered. It is assumed that the methods do not change these. This is a reasonable assumption since, generally, methods in user code cannot modify static fields of library and Java System classes. However, in a few cases such as with the system clock and with I/O devices, this may not be true. This assumption is used in the definition of a method's effects and in decisions on what variables should be tracked in the analysis of effects as described in Section 3.4.2 on page 49.

The second assumption is that the classes in the source code submitted for analysis, have at least one constructor that sets its instance and class variables. This assumption was made because it simplifies the process of generating multiple random objects.

The third assumption is that the *toString* method has been overridden for all of the tested classes. This method is assumed to print out a representation of the object that facilitates the differentiation between structurally different objects. Thus, the assumption is that the *toString* method prints out more than just class name and address.

3.2 Overview

My approach to semantic clone detection using *method IOE-Behavior*, analyzes Java code at the method level to detect functional equality. Recall from Section 2.4.1 on page 21 that two methods are *functionally equivalent* if they have the same input-output and effects behavior: *IOE-Behavior* [3]. There are two contributors to input, namely: the set of parameters passed to the method, and the heap state when the method is invoked. Output is the return value of the method (none for void methods). A method's effects are how it changes the heap, that is, any mutations of non-local variables that persist beyond the method call.

The semantic clone detection algorithm developed in my research is a 4 step process: *abstraction*—line 2, *filtering*—lines 4–9 and 12–24, *testing*—lines 26–27 and *collection*—lines 28–30, as described in the pseudo code in Figure 3.1 and discussed in the sections that follow.

```
1  SemanticCloneDetection(input_files){
2    Create Program AST $P$ from input_files
3    set<methodTypes> $types$
4    for each method $m$ in $P$
5      add $m$.getType() to $types$
6    for each methodType $t$ in $types${
7      if number methods with type $t$ < 2{
8         delete $t$ from $types$
9          }
10   }
11   Map<TypeAndEffect,methods>$tAndE$
12   for each methodType $t$ remaining in $types${
13    $matchingmeths$ = methods with type $t$
14    for each method $meth$ in $matchingmeths${
15      $meth$.getEffects()
16    }
17    Map<TypeAndEffect,methods>$tmp$ = group methods by effect
18    for each TypeAndEffect $e$ in $tmp${
19     if matching methods < 2{
20      delete $e$ from $tmp$
21     }
22    }
23    $tAndE$.add($tmp$)
24   }
25   for each TypeAndEffect group in $tAndE${
26          create test files  $files$
27          run $files$
28          collect clones
29   }
30  output clones
31 }
```

Figure 3.1: Pseudo code for algorithm for semantic clone detection

3.3 Abstraction

The first step is the creation of an abstract representation of the code for analysis. This corresponds to line 2 of the pseudo code in Figure 3.1. The code submitted for testing is parsed to create an abstract syntax tree (AST). This is generated by a static analysis tool built on the JastAdd compiler generator, created by Hedin and Magnusson [8]. This tool (discussed in detail in Section 3.7 on page 70), uses attribute grammars to decorate the AST with synthesized attributes *methodType* and *method effect* for AST nodes. The two attributes are declared as follows using JastAdd attribute grammar syntax:

```
syn HashMap<MethodType,ArrayList<MethodEntry>> MethodDecl.
    MethodTypes();
```

```
syn EffectType MethodDecl.Effects()circular[new EffectType()];
```

They are stored as *MethodType* and *EffectType* objects, which are described in Section 2.3.1 on page 13 and Section 2.3.2.2 on page 18. Equations are available to lazily compute the value of these synthesized attributes for each of the AST nodes.

Figure 3.2 on the next page describes the data structures used to define and store the attributes required to evaluate semantic equivalence. The *MethodTable* is a store of all of the methods identified in the code. *TypeAndEffect* is a record used to store a method's type, (described in Section 2.3.1 on page 13) and its effect, (described in Section 2.3.2.2 on page 18). These records are used to create the equivalence classes of methods with the same IOE-Behavior. *MethodType* defined in Section 2.3.1 is stored as an object with the data members return type and the parameter type list. *Effects* are stored as an *EffectType* object which is an arraylist of Strings of the form *classname.fieldname.* A *MethodEntry* is an object which stores specific method information. The first data member is the full name of the class where the method is declared. *Methodname* is the name of the method. The boolean stores whether the method is static. The last two integers store line numbers for the beginning and ending locations of the method in the code. *Fieldname*, *methodname* and *classname* are string identifiers. Parameters are defined as Accesses which are the JastAdd Java compiler's representation of legal Java types.

The type of a method, (its *methodType*), is computed by extracting the argument and return type information from the method's declaration. This information is obtained by querying the attributes *typeAccess* and *Parameter* of the *MethodDecl* node as shown in the code of Figure 3.3. The *MethodDecl* node has components *ParameterDeclaration** and *TypeAccess* representing the argument types and return type respectively. In my implementation, *methodType* is declared as an object defined by the *MethodType* class which contains 2 private data members:

```
MethodTable = HashMap<MethodTypeEffect, ArrayList<MethodEntry>>
MethodTypeEffect = (MethodType, Effect)
MethodType = (ReturnType, Parameter+)
Effect = EffectType
EffectType = ArrayList<String>
MethodEntry =(Classname, Fieldname, Methodname, boolean, int, int)
Fieldname = id<String>
Methodname = id<String>
Classname = id<String>
Parameter = Access
Access = type<String>
```

Figure 3.2: Data structures used in computing semantic equivalence of methods

```
eq MethodDecl.getMethodTypes(){
        ArrayList<ParameterDeclaration> p = new ArrayList<
           ParameterDeclaration>();
        int size = getNumParameter();
        for(int i = 0; i < size; i++){
                        p.add(getParameter(i));
        }
        Access reType = getTypeAccess();
        return new MethodType(reType, p);
}
```

Figure 3.3: Code to compute methodType

```
          Access returnType;
          ArrayList<Parameter> parameters;
```

My effects analysis is a modification of the type-based, field-based analysis, described by Razafimahefa [48] and explained in detail in Section 2.3.2.2 on page 18. The goal is to identify persistent changes to the heap. Hence, only updates to receiver objects and non-local object variables are recorded.

As described in Section 2.3.2.2 on page 18, the *effect* of a method is stored as a set of Strings, where each String is of the form *classname.fieldname*—where *classname* is the class of the object mutated, or the class to which the static field belongs; and *fieldname* is the field updated. Thus, there is no distinction made between different objects of the same type. For example, in the method *foo* in Figure 3.4, the effects resulting from the update on the two *Acct* objects *a* and *b* are the same— update of the balance field of an *Acct* object.

```
1 int foo(Acct a, Acct b){
2    ...
3    a.balance = 25.00;
4    b.balance = 100.00;
5    ...
6 }
```

Figure 3.4: Example of how method effects are recorded

Both lines 3 and 4 of code have an *effect* of *{Acct.balance}*. Since duplicates are ignored, the net *effect* of method *foo* is recorded as *{Acct.balance}*.

Table 3.1: Calculating a Method's Effects

Code	Statement Effects	Method Effects
double getBalance(){		{}
return balance;	{}	
}		
void updateBal(double newBalance){		{Acct.balance}
balance = newBalance;	{Acct.balance}	
}		
String withdraw (double amt){		{Acct.balance, Acct.lastActivity}
double acctBalance = getBalance();	{}	
if(acctBalance > amt){		
acctBalance-=amt;		
updateBal(acctBalance);	{Acct.balance}	
lastActivity = todayDate;	{Acct.lastActivity}	
return “Withdrawal was successful.”;		
}		
return “insufficient funds for withdrawal.”;	{}	
}		

When computing the effects of a method, each statement is statically analyzed and the set of writes, (that is those found on the left hand side of assignment statements) is collected. This is a ‘*may*’ analysis, since it is an over-approximation of the actual writes, and tracks all of the possible writes. For example, the effect analysis of conditional statements is an over-approximation, since the record of possible writes also includes those within conditional statements, which might not occur at runtime. For method call statements, the effect is recorded as the *effect* of the called method. This is demonstrated in the computation of the effects for the *withdraw* method of the *Acct* class shown in Table 3.1 on the preceding page.

The *effect* of a method is declared as a circular attribute in JastAdd, since there may be mutual recursion among methods. This computation will always terminate since the effect is stored as a set of *classname.fieldname* pairs (both static and instance fields), and there is only a finite number of class-name and field-name pairs in any program. The net effect of a method is evaluated by computing the effects of each of the method's statements and then taking the least upper bound (essentially a union) of that. Thus for a sequential method A, with statements a_1,... a_n, the effects of A are given by equation 3.1.

$$A.getEffects() = \sqcup_{i=1}^{n} a_i.getEffects() \tag{3.1}$$

Computations for the effects of a loop structure are also similar, since duplication is removed from the method effects, the effects of the loop would be the same as the effects of one iteration of the loop.

The code in Figure 3.5 gives the attribute equation for a method's effects. The AST node structure for MethodDecl and Block are given by:

```
MethodDecl : MemberDecl ::= Modifiers TypeAccess:Access <ID:String>
Block : Stmt ::= Stmt*
```

If the method is abstract, then its effect is $\top$ (*Top*). This is a value used to represent an over-approximation of unknown effects. As a result, all abstract methods of the same *methodType* are placed in the same effects group. For non-abstract methods, the effect of the method would be the net effect of the *Block* (The computation is shown in lines 11–21). First, the *effect* of each statement of the method is evaluated. The join of these values is then recorded as the method's net effects. This is explained in greater detail in Section 3.4.2 on page 49.

```
eq MethodDecl.getEffects(){
  ArrayList<String> ret = new ArrayList<String>();
  if(hasBlock()){ //not abstract
  ret.addAll(getBlock().getEffects());
  }
  else{
    return Top
  }
}
//-------------------------------------------------
eq Block.getEffects(){
  int size = getNumStmt();
  ArrayList<String>ret = new ArrayList<String>();
  for(int i = 0; i < size; i++){
    ArrayList<String>tmp = Stmt.get(i).getEffects();
    ret.addAll(tmp);
  }
  HashSet h = new HashSet(ret);//removing duplicates
  ret.clear();
  ret.addAll(h);
  return ret;
}
```

Figure 3.5: Code to compute the effects of a method

Table 3.2 shows the values of the attributes *methodtype* and *effects* for 3 methods the *Acct* class.

Table 3.2: *MethodType* and *effect* attributes for 3 methods

Method	MethodType	MethodEffect

double checkBal(){
  return balance;
}
	double()	{}

void DoubleBal(){
  balance = balance * 2;
}
	void()	{Acct.balance}

String copyBal(Acct a){
  a.balance = balance;
  return “success”;
}
	String(Acct)	{Acct.balance}

3.4 Filtering

All of the methods in the code being evaluated, are identified as a preliminary set of potential method clones—the candidate set. This set is then reduced or refined by applying the *methodType* and *effects* filters sequentially. The remaining candidate set (which is a smaller set than the original one), can then be evaluated further and screened for equivalence. The filtering phase possibly reduces the number of methods that have to be evaluated, thus improving the overall efficiency of the detection algorithm.

3.4.1 MethodType Filter

The first filter, uses the *methodType* attribute described in Section 2.3.1 on page 13, to group methods into equivalence classes. Methods with equivalent *methodType* (that is return type and parameter types are the same), are placed in the same equivalence class. Only equivalence classes containing more than one method are considered for further analysis (see algorithm overview in Figure 3.1 on page 41, lines 4–9). The pseudo code for the *methodType* filter is given in Figure 3.6 on the following page.

The set of *methodTypes* for a given Program, is the union of the *methodTypes* found in each of its classes and/or interfaces. *The ReferenceType.getMethodType()* method in Figure 3.6 on the next page shows the pseudo code for the computation of the *methodTypes* in a class or interface. The algorithm takes each method in the *ReferenceType* in turn, and adds it to a set of methods of matching type. The result is a set of mappings of methodType to methods with that type. The addition of the new method to the set is shown in lines 25–31 of Figure 3.6. For each method to be added, a check is made to determine if its type already exists in the mapping (line 27 of code). If that *methodType* is already in the mapping, the new method is added to the set of methods with that type already in the mapping (line 28 of code). Conversely, if the *methodType* is not yet present, then a new entry is made, that maps the *methodType* of the new method, to the new method (lines 29–32 of code).

```
 1 Set<MethodType> Program.getMethodType() {
 2   Set<MethodType> ret
 3   for each ReferenceType r//(class or interface) in Program
 4      AddTypes(r.getMethodType(),ret)
 5   ret = ret - types with single matching method
 6   return ret
 7 }
 8 ----------------------------------------
 9 Set<MethodType>AddTypes(Set<MethodType>ret,Set<MethodType>mt){
10   if ret is empty return mt
11   if mt is empty return ret
12   for each methodType t in mt{
13     meths = methods with type t in mt
14     find matching methods matching in ret
15     if (matching is not empty)
16       matching += meths
17     else
18      add t → meths to ret
19   }
20   return ret
21 }
22 //----------------------------------------
23 Set<MethodType> ReferenceType.getMethodTypes(){
24   Set<MethodType> ret
25   for each method m in ReferenceType{
26     type of m = t
27     if ret already has t
28      add m to list of methods with type t in ret
29   else{
30     add  t as new MethodType
31     add m as method with that type in ret
32     }
33   return ret
34   }
35 }
```

Figure 3.6: Pseudo code for methodType filter

Table 3.3: Sample methods for methodType analysis

Methods
void deposit(double)
double checkBal()
double calInt()
String getType()
String printState(double)
void withdraw(double)

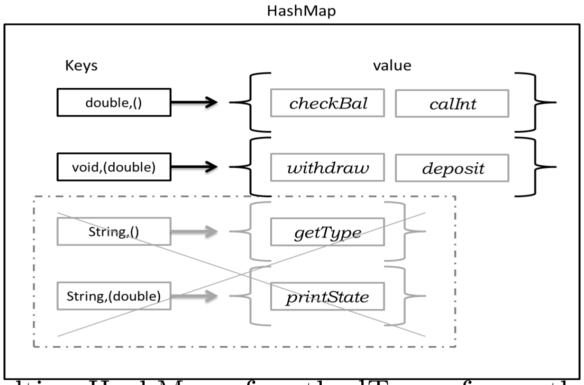


Figure 3.7: Resulting HashMap of methodTypes for methods in Table 3.3

For example, given the six methods in Table 3.3 on the preceding page, applying the methodType filter would result in 2 equivalence classes of candidate clones: {`checkBal, calcInt`} and {`withdraw, deposit`}. The fourth method, `getType`, would be filtered out from the first set, since although it has the same parameter list as `checkBal` and `calcInt`, its return type is different, thus its output behavior is also different. The fourth and fifth methods `getType` and `printState` do not form a candidate set either, because, although their return types are the same, their parameter lists are unequal, indicating a variant in input behavior. Figure 3.7 shows the state of the resulting HashMap for the six methods. The keys with only one *methodEntry* are deleted leaving only two *methodTypes*, each with 2 methods.

3.4.2 Effects Filter

The *effects* filter further refines the equivalence classes returned by the *methodType* filter. The pseudo code is shown in Figure 3.8 on the following page. Beginning with the equivalence classes of *methodTypes* returned by the *methodType* filter, each equivalence class is taken in turn and the *effects* for its members computed lazily. Methods with the same effects are grouped into a new equivalence class (lines 4–10 in Figure 3.8). As a result, the methods in an effect group, have the same *type* and *effect*. This information is stored as a *TypeAndEffect* object as shown in line 7. Each method of an equivalence class is taken in turn and its *TypeAndEffect* is computed. If that key already exists in the effectsHashMap then the value for that key is updated to include the new method. Otherwise, a new entry is made in the *effectsHashMap* to add that method. The pseudo code which handles this, is shown in lines 17–23 of Figure 3.8. As before, any method left in a group on its own, is no longer considered a candidate. Hence, the *effects* filter can have any of the following effects on an equivalence class returned by the methodType filter:

- no change
- delete equivalence class
- reduce the size of equivalence class
- split into smaller equivalence classes

```
 1 Program.getEffects(){
 2   Set<MethodType> Types = Program.getMethodType()
 3   Map<TypeAndEffect> ret
 4   for each  methodType t in Types{
 5     get set of matching methods meths
 6     for each method m in meths{
 7       TypeAndEffect tE = new TypeAndEffect(t, m.effects())
 8       AddTypeEffects(ret, tE, m)
 9     }
10   }
11   ret = ret - typesAndEffects with 1 matching method
12   return ret
13 }
14 //----------------------------------------
15 Map<TypeAndEffect> AddTypeEffects(TypeAndEffect mappings M,
      TypeAndEffect tE, method m )
16 {
17   if(M has tE){
18     get matching methods match
19     add m to match
20   }
21   else{
22     add tE → m to M
23   }
24   return M
25 }
```

Figure 3.8: Pseudo code for effects filter

Continuing from the example in Figure 3.7 on page 49, if the effects filter is applied to the results of the methodType filter, as shown in Table 3.4 and Figure 3.9, only one equivalence class would be left. The equivalence class containing `checkBal` and `calInt` would be split into two smaller equivalence classes each with one method; since the two methods do not have the same effect. These methods would not be considered for further testing since the equivalence class has only one member. However, the equivalence class containing `withdraw` and `deposit` would remain intact, since these methods have the same effect. This equivalence class would go on to the next stage where the two methods will be tested dynamically to determine semantic equivalence.

Table 3.4: Effects of methods returned by methodType filter in Figure 3.7 on page 49

Methods	Code	Effect
`double checkBal()`	`{return balance;}`	`{}`

`double calInt()`	`{ double interest =`
	`balance * 0.05;`
	`balance += interest;}`
		`{Acct.balance}`

`void withdraw(double amt)`	`{balance -= amt;}`	`{Acct.balance}`
`void deposit(double amt)`	`{balance += amt;}`	`{Acct.balance}`

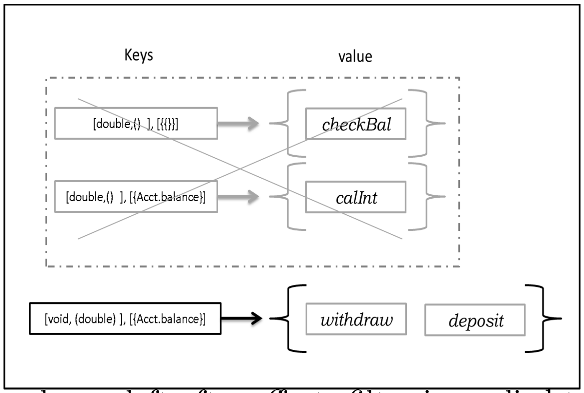

Keys
value
[double,()], [{{}}]
checkBal
[double,()], [{Acct.balance}]
calInt
[void, (double)], [{Acct.balance}]
withdraw
deposit

Figure 3.9: Equivalence classes left after effects filter is applied to methods in Table 3.4

3.4.3 Benefits of filters

The benefit of the two pre-test filters is to reduce the number of methods that require dynamic testing. Without these filters, the number of methods to be tested would increase considerably. In the example in Figure 3.7 on page 49 and Figure 3.9 on the preceding page, not using the filters would require that all 6 methods be tested using the dynamic tests. If only the methodType filter was used, 4 methods would need to be tested (see Figure 3.7 on page 49)—this is a reduction of approximately 33% in the required tests. When both filters are used as shown in Figure 3.9 on the preceding page, only two methods need to be tested dynamically — a reduction of about 67% compared to the number of required tests without the filters. This is an important contribution in terms of the efficiency of the general algorithm, since dynamic testing may be arbitrarily costly. Real data on the efficiency gains in practice are provided in Section 5.3.2 on page 87. There, the reduction in the number of methods for testing ranges from 20% to 48%.

3.5 Testing

The testing phase, shown in lines 25–27 in Figure 3.1 on page 41, is the ultimate test for the candidate clone sets produced by the filters. This phase is a dynamic filter, comparing the actual behavior of methods when called in the same context. The context consists of heap states (the value of receiver object fields and other non-local instance and class fields) and parameter values. The testing phase has 3 sub-phases, namely: *creating the test context*—test objects and test parameters; *generating test files*; and *executing the test files.*

3.5.1 Creating the Testing Context

The testing context for a method is the input as described in Section 2.4.1 on page 21; it is the set of actual values for parameters, fields of receiver objects and static fields. The creation of this context is necessary for running the dynamic tests.

Primitive values are generated by using Java's random number generator in the method *makeParameter(...)* shown in Figure 3.10 on the following page. This method takes two parameters, the type of the primitive parameter or String to be created (eg. "byte", "int", "short"); and a flag indicating if a single parameter is required or an array of that parameter type. This method is also used to generate primitive and String data members and static field values.

The method *randomObjectCreationString()* shown in Figure 3.11 on page 55 is used to generate objects. They are created with a call to a randomly selected constructor of the object's class (see Figure 3.12 on page 55), with randomly-generated parameter values where needed (see Figure 3.13 on page 56). When the parameter is an object, a recursive call to *randomObjectCreationString()* is used to generate the argument object. Since it is practically impossible to test all combinations of inputs and heap states for all methods, a modified stratified random sampling of contexts is used to select a representative sample for test cases. This is discussed in Section 2.6 on page 32.

```
public static final Random randomgen = new Random();
public static final String arrayElementDelimiter = "&";

public static String makeParameter(String type, int size)
   throws ClassNotFoundException
   {
   StringBuilder sb = new StringBuilder();
   if(size > 1){sb.append("new␣" + type + "[]");}
        for(int i = 0; i < size; i++){
     if(type.equals("char")){
      char c = char(randomgen.nextInt(Integer.MAX_VALUE));
      sb.append("\'" + c + "\'");
   }
     else if(type.equals("byte")){
      sb.append(new Integer(randomgen.nextInt(Integer.MAX_VALUE
         ).byteValue());
     }
     else if(type.equals("int"))
      sb.append(Integer.toString(randomgen.nextInt(Integer.
         MAX_VALUE)));
     else if(type.equals("double"))
      sb.append(Double.toString(randomgen.nextDouble()));
     else if(type.equals("float"))
      sb.append(Float.toString(randomgen.nextFloat())+ "f");
     else if(type.equals("short"))
      sb.append(new Integer(randomgen.nextInt()).shortValue());
     else  if(type.equals("long"))
      sb.append(Long.toString(randomgen.nextLong())+ "L");
     else if(type.equals("boolean")){
      int val = randomgen.nextInt(2);
      if(val == 0) sb.append("true");
      else sb.append("false");
     }
     else if(type.equals("String")){
      sb.append("\"" + Long.toString(Math.abs(randomgen.
         nextLong()), 36)+ "\"");
                }
     else{
      String s = randomObjectCreationString(type);
      if(s == null)return null;
      sb.append(s);
     }
     if((size > 1)&& (i < size -1)){
      sb.append(arrayElementDelimiter);
     }
   }
   return sb.toString();
 }}
```

Figure 3.10: Code randomly generate parameters and objects

```
public static String randomObjectCreationString(String classname)
  throws ClassNotFoundException
  {
  Random randomgen = new Random();
  String s;
  Constructor[] constrs;
  try{
  Class c = Class.forName(classname);
  Class enclosing = c.getEnclosingClass() ;
  if(enclosing == null){
    if(Modifier.isStatic(c.getModifiers()))
      return "new␣" + classname + "()";
    else {  //if(isInnerClass())
      return "(new" + enclosing.getName() + "()).new␣" + classname + "()";
    }
  }
  return getConstructorString(c);
  }//end try
  catch(ClassNotFoundException e){
   return null;
  }
}
```

Figure 3.11: Code for method *randomObjectCreationString*

```
String getContructorString(Class c) throws ClassNotFoundException{
  StringBuilder sb = new StringBuilder();
  Constructor[] constrs = c.getDeclaredConstructors();
  int size = constrs.length;
  if(size  > 0){
    int pos = randomgen.nextInt(size);
    Constructor choice = constrs[pos];
    Class[] params = choice.getParameterTypes();
    //constructor has no parameters
    if(params.length == 0)
      return "new␣" +  classname + "()";
    else{
          StringBuilder sb = new StringBuilder();
          sb.append("new␣" +  classname + "(");
          String cps = getConstrParams(params);
     if(cps == null) return null;
     else sb.append(cps + ")");
     return sb.toString();
    }
  }
}
```

Figure 3.12: Code for randomObjectCreationString helping function ConstructorStr

```
public static String getConstrParams(Class[] params)
  throws ClassNotFoundException
  {
  StringBuilder sb = new StringBuilder();
  String name,s;
  int arrayIndicator;
  for(int i = 0; i < params.length; i++){
    name = getMyName(params[i]);
         if(params[i].isArray())
           arrayIndicator = 2;
         else
           arrayIndicator = 1;
         s = makeParameter(name,arrayIndicator);
    if(s==null)
         return null;
         sb.append(s);
    if(i == params.length-1)
           sb.append("\)");
         else
           sb.append(",");
   }
   return sb.toString();
 }
```

Figure 3.13: Code for helping function getConstrParams

Interfaces and abstract classes present a challenge since they cannot be instantiated. An instantiable subtype is used instead. This is achieved using Java *Instrumentation*. The Instrumentation API available in Java versions from 1.5, allows for querying the set of classes loaded by the JVM for a set of subtypes of the required interface or abstract class (this is described in more detail in Section 2.8 on page 37).

A user-defined number of test cases (*numChoices*) is used to determine how many such subtypes will be collected before one is selected at random. If less than *numChoices* possible subtypes are found, then the list is limited to the number found. One subtype is selected at random from the retrieved list. The methods *getInterfaceSubType* shown in Figure 3.14 on the following page and *getClassSubClass* in Figure 3.17 on page 59 are used to find the subtype of an interface and the sub-class of an abstract class, respectively.

This process of querying the JVM using instrumentation can be computationally costly, so the algorithm maintains a cached list of all already identified subtype substitutions to reduce the cost of subsequent searches. This list is updated each time a new substitution is made (see Figure 3.15 on the following page). This makes the creation of abstract objects and interfaces less random, but reduces the cost of the computation. This trade off was accepted since the cost of instrumentation is very high (about 0.2 seconds).

While the selection of the subtype would be less random, the constructor used to create the test object is still selected at random; thus maintaining some level of randomness. In cases where no class substitution can be found, the affected methods are not tested, since their method call cannot be constructed. However, the other equivalence classes may still be tested. A message is also output to the user that a substitute could not be found for the named class.

```
final Random r = new Random();
public static Class getInterfaceSubType(Class iface, int
    numChoices){
  //Input: accepts an interface name, and the number of options
        to explore
  // searches the list of loaded classes for classes
      implementing the interface
  //Output: returns a randomly selected member of the first n
      sub-types found          //where n = numChoices
   int pick = r.nextInt(numChoices);
  HashMap<Class, ArrayList<Class>> currentTable =
      MyJavaAgent.getImplementingTable();//cached list
 //search cached list
  Class c;
  c = searchAlreadyFound(currentTable,iface,numChoices,pick);
  if(c!= null){
        return c;
  }
  else{
        c = getLoadedSubType(iface,numChoices,pick);
  }
  return null;
}
```

Figure 3.14: Code to obtain interface sub-type

```
Class searchAlreadyFound(HashMap<Class, ArrayList<Class>> currentTable,Class iface, int numChoices, int pick){
  if(currentTable.containsKey(iface)){
        ArrayList<Class> tmp = currentTable.get(iface);
        if(tmp.size()< numChoices){
           pick = r.nextInt(tmp.size());
        }
    return tmp.get(pick);
  }
  else
   return null;
}
//----------------------------------------------------
Class getLoadedSubType(Class iface, int numChoices, int pick){
  //search for implementing classes in loaded classes
  Class[] cls = MyJavaAgent.getInstrumentation().getAllLoadedClasses();
  int numfound = 0;
  int iter = 0;
  ArrayList<Class> targetClasses = new ArrayList<Class>();
  while((iter < cls.length) && (numfound < numChoices)){
        Class curr = cls[iter];
        if(Arrays.asList(curr.getInterfaces()).contains(iface) && !curr.isInterface()&& !isAbstract(curr)) {
          targetClasses.add(curr);
          numfound++;
        }
        iter++;
  }
  if(!targetClasses.isEmpty()){
        updateImplementingTable(iface, targetClasses);
        if(targetClasses.size() < numChoices){
          pick = r.nextInt(targetClasses.size());
        }
        return targetClasses.get(pick);
  }
  return null;
}
```

Figure 3.15: Helping methods for finding interface subtypes

```java
Class getLoadedSubClass(Class SpClass, int numChoices, int pick){
  //search for sub-classes in loaded classes
  if(!MyJavaAgent.getInstrumentation().equals(null)){
    Class[] cls = MyJavaAgent.getInstrumentation().getAllLoadedClasses();
    int numfound = 0;
    int iter = 0;
    ArrayList<Class> subClasses = new ArrayList<Class>();
    while((iter < cls.length) && (numfound < numChoices)){
      Class curr = cls[iter];
      if((SpClass.isAssignableFrom(curr)) && !isAbstract(curr)){
        subClasses.add(curr);
        numfound++;
      }
      iter++;
    }
    if(!subClasses.isEmpty()){
      nonAbstractSubClassTable.put(SpClass, subClasses);
      if(subClasses.size() < numChoices){
        pick = r.nextInt(subClasses.size());
      }
      return subClasses.get(pick);
    }
         else{
      return null;
    }
  }
}
```

Figure 3.16: Code to search loaded classes for sub-class of abstract class

```java
public static Class getClassSubClass(Class SpClass, int numChoices){
 Random r = new Random();
 Class c;
 int pick = r.nextInt(numChoices);
 HashMap<Class, ArrayList<Class>> currentTable = MyJavaAgent.getSubClassTable();
 //check Cache
 c = searchAlreadyFound(currentTable,SpClass,numChoices,pick);
 if(c!= null){
   return c;
 }
 else{
   c = getLoadedSubClass(SpClass, numChoices, pick);
   if(c!= null){
     return c;
   }
 }
 return null;
}
```

Figure 3.17: Code to obtain subclass for abstract class

3.5.2 Generating the Test Files

The generation of the test files is carried out through an automated process described in the pseudo code in Figure 3.18. The main generated test file consists of a series of method calls to the methods in a given equivalence class. It is my hypothesis that a good approximation for detecting semantic equivalence between methods can be obtained by running these methods on a sufficiently large sample of their input domain—including both state and argument values.[1]

```
Files generateTestFiles(int numTests){
  HashMap<TypeAndEffects, ArrayList<MethodEntry>>equiClasses =
      Program.getTypeAndEffect();
  For each TypeAndEffects mapping in equiClasses{
     Generate files for class
  }
  create driver class to run all files
}
```

Figure 3.18: Pseudo code for generating test file

To test this hypothesis, my algorithm runs multiple tests on objects in the same state and also runs each set of input parameter values with objects in multiple states. The details of how test states or contexts are created are described in Section 3.5.1 on page 53. Each method is tested with a user-defined number of test cases *numTests*. This is called the test set. For methods with a non-empty parameter list, *numTests* sets of parameter values are used. For non-static methods *numTests* receiver object states are used. For static methods, *numTests* tuples of the static fields of the class containing the method are used if such fields exist. The structure of the test file is thus determined by the *methodType* of the equivalence class as summarized in Table 3.5 on page 63. Pseudo code for handling cases with non-empty and empty parameter lists are given in the first and second methods in Figure 3.20 on the following page.

[1] The algorithm uses a timeout for method executions.

```
generateFilesForEquiClass(Equivalence class of methods grouped
    by type and effect class){
   Create class with constructor for equivalence class
   if methodType has parameters
      generate numTests sets of parameters
   if has non-static methods
      generate numTests sets of each receiver object
   if has static fields
      generate numTests sets of static fields
   for i=1 to number of methods in class{
     if(has parameters ){
            getMethodCallStringsWithParameters(method,
               parameters, states)
     }
     else{
            getMethodCallStringsWithoutParameters(method,
               states)
     }
   }
   create aspect code using method call strings
   write aspect code to files
}
```

Figure 3.19: Pseudo code for generating test files for an equivalence class

```
String getMethodCallStringsWithParameters(method, parameters,
    states){
    for i =1 to numTests{
          for j =1 to numTests{
            create method call with state i and parameter set j
                add code to store result as EffectRec in array
      }
          add code to Convert result array to String  and hash
             into result hash on resultString
        }
}
//---------------------------------------------------

String getMethodCallStringsWithoutParameters(method, states){
    for i =1 to numTests{
          create method call with state i
          add code to store result as EffectRec in array
    }
          add code to Convert result array to String  and hash
             into result hash on resultString

}
```

Figure 3.20: Pseudo code for generating method call strings

For example, suppose the user-defined *test-set* size is 5. Then to test a non-static method $A.b(c, d, e)$ we use 5 samples of objects of type A in different states. We also use 5 different tuples of the parameters c, d, and e. Note that this is only 5 tuples, not all combinations of 5 values for each parameter. Thus the total number of tests for method b is 5×5 or 25.

Analyzing private methods to identify semantic clones presented a challenge: how to make calls to private methods outside of their class, when they are only visible inside of their defining class. I considered three options for handling this. The first was to modify the code being tested, so that all methods are declared public. A second approach, like the first, would require modifying the code in the input test classes. In this approach new public methods can be defined within a class, to call the private class methods. A third possible approach is to generate privileged AspectJ aspects. This is described in Section 2.7 on page 34. This approach presents several advantages. It facilitates the desired functionality with very little implementation effort, since existing software is reused.[2] Plus there is no need to change the code in multiple input files. Also, the use of a privileged aspect also allows access to private methods and fields without changing their visibility. Thus the AspectJ approach is adopted in this study.

For each equivalence class of candidate clones, a matching class and at least one privileged aspect are created. The class is used to instantiate the instance of the equivalence class so that its member methods can be run. The privileged aspect defines the *runTest* method. This is fundamentally a set of method calls to each of the methods of the equivalence class in all of the created contexts. The *runTest* method is defined as an advice, when the pointcut of a call to an equivalence class constructor, is encountered. The method *runTest* is placed in a privileged aspect to ensure that calls to private methods, would not be blocked

[2] Actually AspectJ is implemented using the second approach.

by Java's visibility rules. A single test-driver is also created. This is the file used to run all of the tests. It creates an instance of each equivalence class, running its *runTest* method as part of the equivalence class constructor. Thus it runs all of the methods under test to identify the clones.

Table 3.5: Structure of Test File

MethodType		Structure of Test File
Empty Parameter List	Non-Static	n method calls to each method, each time with a different receiver object state
	Static	if member class has static fields, n method calls to method, each time with a different static field values; Otherwise one method call to method
Non-Empty Parameter List	Non-Static	n receiver objects are created, and n tuples of parameter values. In all $n \times n$ method calls are made to method. Method is called on each receiver object with each of the parameter tuples in turn.
	Static	n tuples of parameter values are created; if member class has static fields n tuples of static field values are created. If there are static fields $n \times n$ method calls are made to method. For each tuple of static field values, the method call is made with each of the parameter tuples in turn. Otherwise the method is called n times – once with each tuple of parameter values

3.5.3 Running the Test Files

The file generation stage generates at least 2 files for each equivalence class—a class file and at least one privileged aspect. The structure of the equivalence class is shown in Figure 3.21 on the following page.

The generated test files evaluates the dynamic behavior of the clone candidates, by using method calls to run each member of an equivalence class on the same input and then comparing the corresponding *outputs* and *effects*. Part of the dynamic testing for semantic clones, is determining equivalence. The testing phase compares methods in a candidate set for identical IOE-Behavior. The comparison of output values is trivial, for primitive types. However, for values that are objects, their final states are checked for *structural equivalence*.

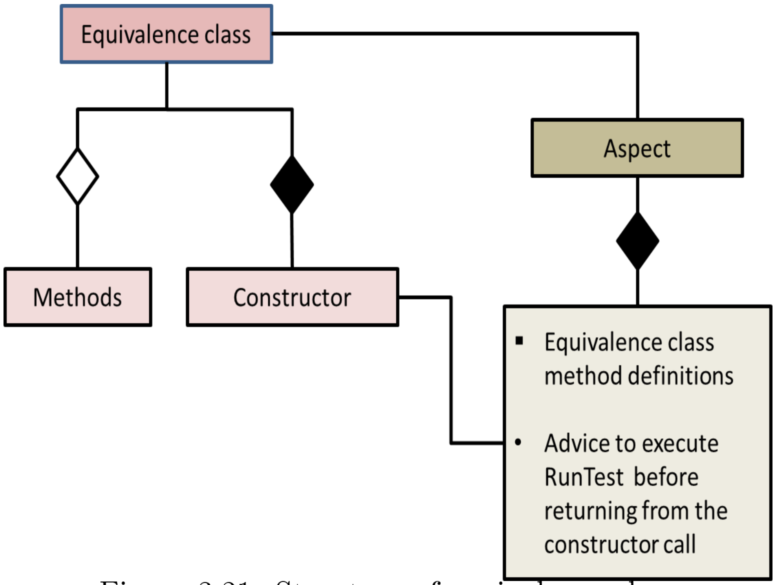

Equivalence class
Aspect
Methods
Constructor
▪ Equivalence class method definitions
• Advice to execute RunTest before returning from the constructor call

Figure 3.21: Structure of equivalence class

To perform structural equality tests, I use the *newEquals* method shown in Figure 3.22 on the next page. This method uses a depth first traversal and compares objects by recursively comparing the corresponding fields. To prevent the comparison from entering an infinite loop for circular objects, a *timeout* feature is used to terminate the comparison at a certain depth of the search. If the values are the same up to that point, the two objects are considered clones. The limitations introduced as a result, are discussed in Section 6.4 on page 107.

To facilitate this comparison of method outputs, the results of the method tests are stored in an EffectRec object. The datamembers of the EffectRec object are shown in Figure 3.23 on the next page.

All of the fields of the EffectRec object are Strings. Thus the components of the output of a method's execution need to be converted to Strings. Strings are used because they are easy to compare. Other types of objects were not used because they could be mutated. To obtain the String for primitive values, the value is converted to an object of the corresponding wrapper class and the *toString()* method is used to obtain a String.

```
boolean newEquals(Object o1, Object o2, int time){
 if (time == timeout)
  return true
 if type o1 != type o2
   return false
 current fieldIndex = 0
 while has more fields && time < timeout{
   if(isPrimitive o1.current field){
      if(o1.current != o2.current)
        return false
   }
   else{
     newEquals(o1.current,o2.current, time++)
   }
   fieldIndex ++
 }
 return true;
}
```

Figure 3.22: Pseudo code for newEquals method to test structural equivalence of objects

```
public class EffectRec {
        private String fromretVal;
        private String fromTarget;//from receiver object
        private ArrayList<String>fromParams;
        private ArrayList<String> sfields;
   ...
}
```

Figure 3.23: Structure of EffectRec

The *fromretVal* field is the value returned by the method, represented as a String. The *fromTarget* field represents the state of the receiver object after the execution of a non-static method. For static methods it is by default the empty String. The *fromParams* field keeps track of the state of object parameters after the execution of the method. If there are no object parameters for the method, then this field is set to the empty ArrayList. Otherwise the String representation for each object parameter in order of appearance in the parameter list, is added to the ArrayList to create the value for the *fromParams* field. The *sfields* is the String representation of the state of the static fields (included in the method's effects), after the execution of the method. To create this String an alphabetical listing of all of the static fields that might be modified is created. A String representation for each field in order is added to the ArrayList.

3.6 Collection

This phase involves running the Test Driver created in the generate test files step of my algorithm. As each test file is executed by the Driver, a subset of the methods tested is returned as an equivalence class of semantic clones—that is, each element of the class, produced identical results for every test case. Figure 3.24 shows how clones output are collected by the driver.

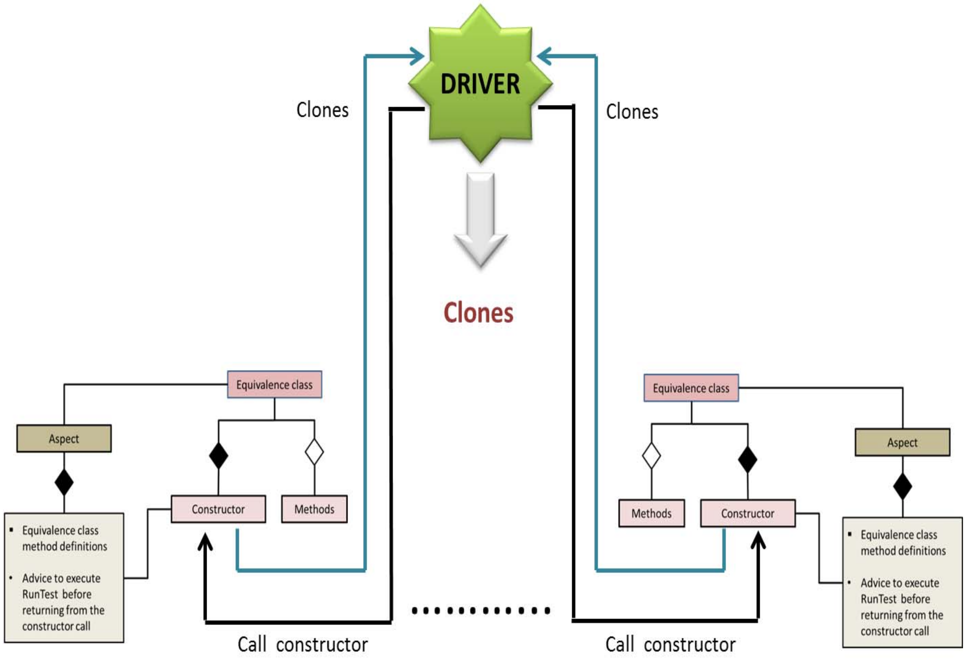

DRIVER
Clones
Clones
Clones
Equivalence class
Aspect
Constructor
Methods
Equivalence class method definitions
Advice to execute RunTest before returning from the constructor call
Equivalence class
Aspect
Methods
Constructor
Equivalence class method definitions
Advice to execute RunTest before returning from the constructor call
Call constructor
Call constructor

Figure 3.24: Running test files with driver

Execution of the test files for an equivalence class produces a 1-dimensional or 2-dimensional array of *EffectRec* for each method, depending on the *methodType* of the methods of the given equivalence class. For *methodTypes* with an empty parameter list, a 1-

dimensional array of *EffectRec* is created for each method as shown in Figure 3.25. This array is of length *numTests*—the user defined number of tests. Each i^{th} index in the array is the result obtained by running the method using the i^{th} generated context as input. For *methodTypes* with a non-empty parameter list, a 2-dimensional array of *EffectRec* is created as shown in Figure 3.26. Each i^{th} row in the array is the result of running the method on the i^{th} generated context. Each j^{th} element in the row is the result of running the method using the j^{th} generated parameter tuple.

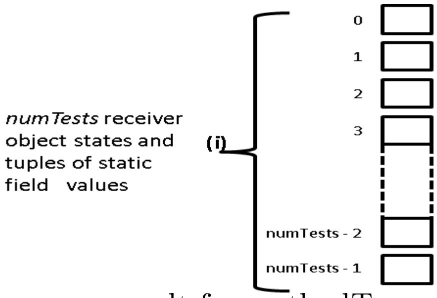


Figure 3.25: EffectRec array result for methodTypes with no parameters

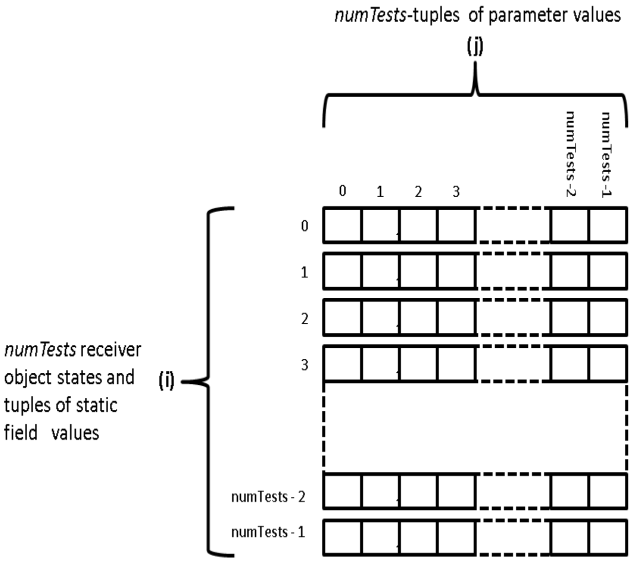


Figure 3.26: EffectRec array result for methodTypes with parameters

Each array is converted to a single String using a combination of the methods *EffectRec.toString()* and the built-in Java method *Arrays.toString()*. For the 1-dimensional array, the *EffectRec* at each index is converted to a String using *EffectRec.toString().* The Strings are then concatenated to produce a single String. For the 2-dimensional array, each row is converted to a String using the same algorithm as the 1-dimensional array. The resulting Strings are then concatenated (in ascending order of row number), to create a single String. The tested method is then added to a Hashmap of clones, on the key of the result-string. The use of the hash table automates the comparison of the method's results and effects; since methods with the same *IOE-Behavior* will produce the same results and therefore the same result string; and will consequently hash to the same location in the Hashmap. When a value hashes to an already occupied row of the Hashmap and the Strings match, the value field of that row is updated to include the new method.

The collection of the semantic clones is done through iteration through the Hashmap. For each key the corresponding value, which is a set of methods is extracted. The values containing more than one method are returned as equivalence classes of clones.

A statistical data report is also output. This includes the following information: number of clones, number and size of clone classes, clone location, clone size in LOC and total execution time as shown in Figure 3.27.

```
                        Cloneclass 0:
------------------------------------------------------------------
org.apache.commons.lang3.concurrent.BasicThreadFactory.
   getThreadCount Location: lines 191 - 193, Clone size: 3 LOC

org.apache.commons.lang3.concurrent.TimedSemaphore.getPeriod
   Location: lines 372 - 374, Clone size: 3 LOC
------------------------------------------------------------------
                        Cloneclass 1:
------------------------------------------------------------------
org.apache.commons.lang3.concurrent.TimedSemaphore.shutdown Location
   : lines 254 - 268, Clone size: 15 LOC

org.apache.commons.lang3.concurrent.TimedSemaphore.acquire Location:
    lines 292 - 310, Clone size: 19 LOC

org.apache.commons.lang3.concurrent.TimedSemaphore.endOfPeriod
   Location: lines 414 - 420, Clone size: 7 LOC
------------------------------------------------------------------
Number of Clones: 5
Number of Clone Classes: 2
Maximum Clone Size : 19 LOC
Maximum Clone Class Size : 3 clones
Minimum Clone Size: 3 LOC
Minimum Clone Class Size: 2 clones
Average Clone Size: 9.0 LOC
Average Clone Class Size: 2.0 clones
Length of program execution: 0.203 seconds
```

Figure 3.27: Sample Output

3.7 JSCTracker

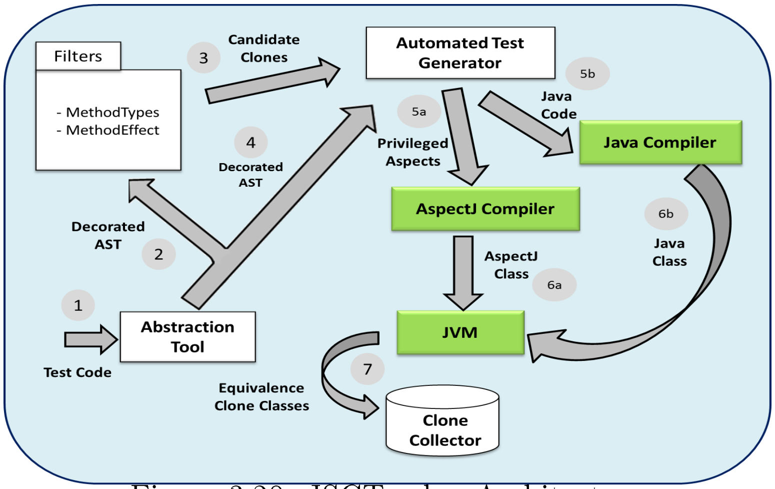


Figure 3.28: JSCTracker Architecture

My semantic clone detection tool—*JSCTracker* is developed to automate the semantic clone detection process using method *IOE behavior*. Its architecture, use cases and level 1 dataflow diagrams are shown in Figure 3.28, Figure 3.29 on page 72 and Figure 3.30 on page 72. *JSCTracker* consists of 5 major components: a *static analysis tool*, a *filtering component*, an *automated test generator*, *compilers* and a *clone repository*. Each of these components is responsible for executing one of the use cases shown in Figure 3.29.

The static analysis tool is used to create the decorated AST for the code to be analyzed. This component uses some of the features of JastAdd [8]; and houses the implementation of the abstraction phase of the CD algorithm as described in Section 3.3 on page 41. This component is directly linked with the user interface to receive the test code. Its output is a decorated AST. The decoration or annotations are generated by 2 files *methodTypes.jrag* and *effects.jrag* which define aspects for computing the program attributes *methodType* and *effects*. Each attribute is defined by synthesized attribute equations, which are essentially methods used to compute the attribute value for the different types of nodes in the AST. The pseudo code for computing these attributes is given in Section 3.3 on page 41. The tool data flows of this component are reflected in processes 1.1 and 1.2 in Figure 3.30 on the next page.

The *filtering component* implements the *methodType* and *effects* filters of the algorithm shown in Figure 3.28 on the preceding page and explained in detail in Section 3.4 on page 47. It accepts the decorated AST as input (as shown in step 2 of the architecture), and outputs sets of candidate clones shown in step 3. The data flow of this component is reflected in process 1.3 of Figure 3.30 on the following page. The filtering mechanism is effected by 2 main files: *TypeAndEffect.jrag* and *Hash.jrag*. The first file defines the attribute *typeAndEffect* for AST nodes. This is the combination of *type* and *effect* of methods. The *Hash.jrag* file defines hash functions for the objects used to store the new program attributes. These functions are necessary, because during the processing of the filters, the program attributes are stored in Java HashMaps as described in Section 3.4 on page 47.

The *automated test generator* accepts the decorated AST and a set of candidate clones as input, as shown in steps 3 and 4 of Figure 3.28 on the preceding page. It queries the decorated AST via attribute accessor method calls, to obtain information about the methods in the candidate clone sets, so as to generate the test files for these methods. The output of is a set of files. A Java class file containing a constructor used to instantiate the candidate clone class and one or more privileged aspects containing the code to call the methods in the candidate clone set. This component of the tool, implements the '*generate test file*' phase of the CD algorithm. It's dataflows are represented by process 1.4 in Figure 3.30 on the next page.

The *compilers* and *JVM* are external components of the tool. They are used to process output files from the *automated test generator*, shown in steps 6a and 6b of Figure 3.28 and process 1.5 of Figure 3.30. Two compilers are used. The Java compiler (JDK 1.7), is used to compile and run the Java class files. The AspectJ ajc compiler 3.0 is used to compile the aspects.

The final component is the clone repository. As the test files for each equivalent set of candidate clones are run, the detected clones are stored in the repository to be output at the end. This is shown in step 7 of Figure Figure 3.28 on page 70.

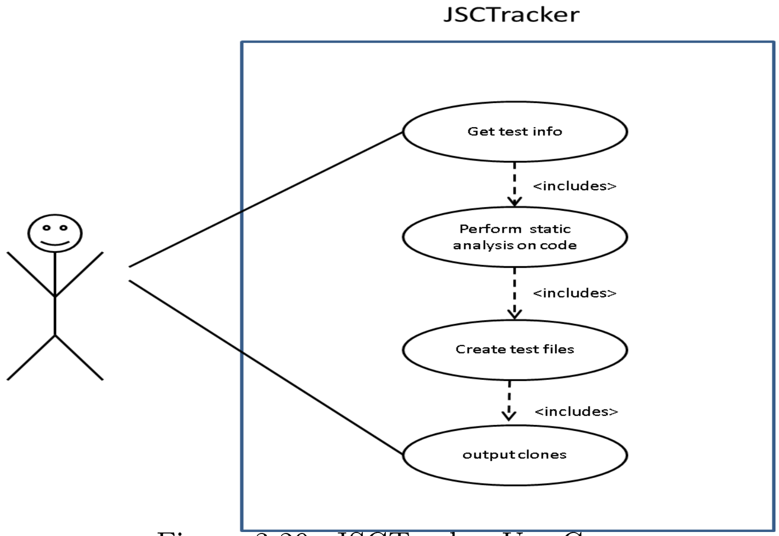


Figure 3.29: JSCTracker Use Case

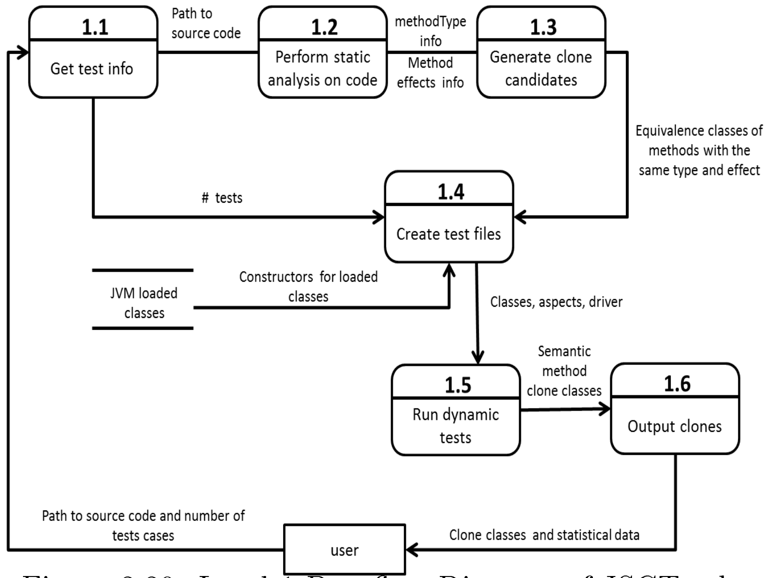

1.1
Get test info
Path to source code
1.2
Perform static analysis on code
methodType info
Method effects info
1.3
Generate clone candidates
Equivalence classes of methods with the same type and effect
# tests
1.4
Create test files
JVM loaded classes
Constructors for loaded classes
Classes, aspects, driver
1.5
Run dynamic tests
Semantic method clone classes
1.6
Output clones
Path to source code and number of tests cases
user
Clone classes and statistical data

Figure 3.30: Level 1 Dataflow Diagram of JSCTracker

JSCTracker takes 5 inputs. Each input is described below.

1. an integer indicting the type of analysis required. There are two options. The number 1 is used for clone detection using only the *methodType* filter. The number 2 is for clone detection using both *methodType* and *effects* filters.

2. an integer, identifying the percentage of similarity between semantic clones detected. To detect semantic equivalence, 100% is used.

3. an integer indicating the size of the *test-set*

4. a String for the path where generated files should be placed

5. a String for the path where the code to be evaluated can be found.

CHAPTER 4
RELATED WORK

Recently, there is increased interest in the detection of semantic clones. Although called by different names: wide-miss clones, high-level concept clones [52], functionally equivalent code [1, 2], behavioral clones [53], representationally similar code fragments [54] and *simions* [44], the goal is the same however, to identify clones created by activities other than copy and paste.

This chapter discusses the current research in semantic clone detection. It outlines the differences between my algorithm and those of other researchers and their relative merits.

4.1 Research on Semantic Clones

Marcus and Maletic [52] present some work on the manual identification of semantic clones which they refer to as wide-miss or high-level concept clones. For the detection process, they use latent semantic indexing, using Mosaic. Their experimental results reveals that their algorithm is not precise, and is unable to identify semantic clones unless all of the identifiers are the same (much like syntactically equivalent clones). My algorithm has precision values of 68% (see Section 5.3.4 on page 94) and is capable of identifying semantic clones with similar or completely different structure and identifier names.

Jiang and Su [2] investigate functional similarity in code fragments within methods in the Linux system using their tool EQMiner. Their algorithm detects semantic clones, using code fragment input-output behavior. They report that about 42% of the semantic clones detected were also syntactic clones, thus the other 58% would be missed by syntactic clone detectors. Jiang and Su's algorithm is scalable, capable of analyzing millions of LOC. This is achieved by using parallelism and some other heuristics. In general they use less restrictive rules in their analysis to foster scalability. However, some of these decisions while improving scalability, may negatively impact the accuracy of the results. I cannot compare the results of my work to theirs, since I study Java code, while they focus on the Linux system written in C. Also, my level of granularity is the method, while theirs is a code fragment. However, the two algorithms can be compared in principle.

For example candidate clones are not compared to all other candidates in their equivalence class. Instead, one member of the equivalence class is selected to represent the group. Each member of the equivalence class is only compared to the representative. While this reduces the number of comparisons from order n^2 to n (where n is the size of the equivalence class), the accuracy of the results depends on the merit of the selected representative. This can lead to false negatives. In my algorithm, the use of the Hashmap to store clone candidates on *methodType* and *effects* keys, implicitly compares each candidate to all of the other members of the equivalence class, thus reducing the probability of returning *false negatives*.

Jiang and Su identify semantic clones as displaying the same input-output behavior. My algorithm also subscribes to this approach and takes it further, to include a method's effects. Effects are not used by Jiang and Su. Another difference is the number of dynamic tests used. They use 10 tests. This improves the time and scalability of the algorithm, but my experiments showed that 10 was an insufficient number of test cases to differentiate between clones accurately. As a result, my tool *JSCTracker* allows the user to enter the number of tests as an input parameter. The user can enter 10–20. However, I recommend 20–30 tests. Increasing the number of test cases explores a wider range of the input domain of the methods. Consequently, the probability of false positives is reduced. Also, since the level of granularity of my investigation is a method, which is a naturally occurring executable program unit, my computations are simple and provide a less invasive and possibly a less disruptive starting point for code maintenance through refactoring. Code fragments on the other hand, need to be modified to create methods with inputs. This is not a trivial process.

Jiang and Su provided no recall value, so my algorithm's recall values could not be compared. However, they do report the same precision value as my algorithm: a false positive rate of 32% which yields precision value of 68%. However, this was computed using half a percent of the returned clusters in some cases. For example in their results for clusters of sizes 2–4 where there were over 10000 found, only 50 were examined). Also, the examined clusters were not selected at random. In my algorithm at least 50% of all returned clones were evaluated in the measure of precision and those examined were selected at random (see Section 5.3.4 on page 94). This difference affects the reliability of the precision values reported.

Another application of semantic clone detection is demonstrated by Kawrykow and Robillard [4]. By analyzing the byte code of libraries and the source code of the library clients, they use functional similarity to identify instances of less than optimal usage of APIs, where developers re-implemented library functions, instead of making calls to an available API function. This project is not defined as a clone detection project, however, conceptually it is an application of semantic clone detection between a project and some selected APIs. Using their detection tool iMaus, they study 10 widely used Java projects from the SourceForge repository. The output of their tool is a set of semantic clone candidates, these are grouped into API usage pattern groups. Each usage group is then manually inspected, to identify and validate the clones found. The tested projects range in size from 20 to 539 KLOC. They find 4–341 method imitations (functional duplicates), with the average precision of 31%.

Kawrykow and Robillard's work has one advantage over mine. They are able to analyze the byte code of the API libraries. However, their precision values are 37% lower and there is no measure of the recall values of the algorithm. Their detection process is not fully automated, hence it is less objective than my automated IOE-Behavior method. Also their work compares the tested projects to API libraries and not to themselves.

Juergens and Gode [44] investigate commercially used Java code *JabRef* to detect semantic clones, using the code fragment as the level of granularity. Through manual inspection of 2,700 LOC, they find that 32 out of 86 utility methods are partially semantic clones — a little over 37%. In their study, they search for semantic clones within JabRef and also between JabRef code and Apache Commons methods. My approach to semantic clone detection differs from theirs in two main ways. Firstly, my algorithm is automated, which makes the semantic clone detection process objective. Secondly, I focus on the behavior of entire methods, while they study code fragments (lines of code that may not constitute a whole method). I select the method as my level of granularity, because of the natural progression that it offers for code refactoring. Also, extracting a code fragment in a way that preserves its semantics, is difficult. In addition, Java does not have call by reference, thus it is difficult to extract variable assignment fragments.

The work presented by Dissenboeck *et al* [1] was developed about the same time as mine, although we were unaware of each other's work. They present an algorithm for the dynamic detection of functionally similar code fragments in Java. Their work focuses on different levels of granularity of code - fragments of methods and whole methods. They also subscribe to the use of input-output behavior through dynamic testing to categorize functionally similar code as described in [25].

My approach to semantic clone detection is different from that defined by Deissenboeck *et al.* [1] because I only consider semantic method clones and include method effects in the detection process. The reason for this choice has already been explained earlier. Another difference is that while Deissenboeck *et al* use all of the constructors for the methods found, to generate the test cases, I use a user-defined number of randomly selected constructors with the same number of randomly-generated actual input values (including object states and heap states). Consequently, while my algorithm may not test every constructor of a class, it produces and tests a larger subset of the input domain; since it also includes a variety of parameter values with each object state and also heap states. My algorithm also combines static analysis to pre-filter the candidate set, before the dynamic testing, thus reducing the actual number of methods to be tested—hence improving the overall efficiency of the clone detection process.

McMillan *et al.* [42] use the concept of semantic clones to detect similarity between Java software applications. In their work, they use a tool *CLAN* (closely related applications) to generate a similarity index from 0–1 between the test subjects. Their research is different from most other semantic clone work because of the granularity of the candidate clones. In most semantic clone detection work, such as mine, the level of granularity is the code fragment or method. However, McMillan *et al.* study the application as a whole. Their work has applications in plagiarism detection and facilitating software reuse.

Keivanloo *et al.* use a metric-based algorithm to detect Type III clones in Java byte code [45]. Their algorithm is implemented in a tool called *SeByte* which uses a combination of set theory and pattern matching supported by the Semantic Web inference engine, in the detection process. One obvious advantage of their algorithm is that they do not need access to the source code. However, while they define their algorithm as semantic clone detection, by the definition used in my research, it is not. Their target is Type III clones which are near miss or parameterized clones (discussed in Section 2.2 on page 9) and these are not necessarily semantically equivalent. For example, two code fragments which are only different in the name of the method that they call, are parameterized clones. However, they would produce different output, unless the two methods had identical effects.

CHAPTER 5
EVALUATION

This chapter describes the processes used to validate my semantic method clone (*SMC*) detection algorithm and the tool—JSCTracker. It also details the case study used as part of this process, and the analysis of the results data obtained. The validation objectives are expressed as seeking answers to the following 4 research questions:

1. Can method IOE-Behavior analysis be used in practice to detect SMCs in Java source code?
2. Do the pre-testing filters used in method IOE-Behavior analysis improve the precision and efficiency, of the SMC detection process?
3. How does SMC detection using method input-output behavior (obtained from method-Type information), compare to that of SMC detection using method IOE-Behavior?
4. How reliable is method IOE-Behavior for the detection of SMCs?

These questions are addressed in a case study, preceded by a pilot test. Each of these is described in detail in the following sections.

5.1 Pilot Test

The first phase of the case study is a pilot test in which JSCTracker is used to automate the SMC detection process. The test subjects are small Java projects including

22 classes ranging in size from 70 lines of code (LOC) with 12 methods, to 900 LOC and 39 methods. These projects are used to test the algorithm's robustness for identifying different categories of SMCs: from the syntactically identical to the very structurally divergent yet functionally equivalent code. For the latter type of clone, I translate C code found in *Hacker's Delight* [6] to create Java methods that are syntactically very different but are semantically equivalent. An example of this type of SMC pair is shown in Figure 2.3 on page 13. The test file containing these methods is *HDelightCode.java*. It can be found in the Appendix , along with the results file. Figure 5.1 on the next page and Figure 5.2 on page 83 show code for a sample of the pilot test files *Calculator.java* and the results from analyzing this class. All of the other test files and corresponding result files are in the Appendix .

Special care is taken in creating the pilot test files, to ensure that both static and instance methods are investigated. Examples of such methods, with a variety of primitive and object return types and parameter types, and arrays of the same are included in the test files to ensure that the different types of SMC are evaluated. The final stage of the pilot test is to evaluate the scalability of the tool, by reflectively evaluating the semantic clones in the source code of JSCTracker.

```
package genfiles;

import java.awt.Rectangle;

public class Calculator {

        private int balance;

        public Calculator(){balance = 0;}
        //-----------------------------------------
        public Calculator(int arg){balance = arg;       }
        //-----------------------------------------
        public int getBal(){return balance;}
        //-----------------------------------------
        public Rectangle OutlineCalculator(){
                return new Rectangle(balance, 2*balance);
                          }
        //-----------------------------------------
        public Rectangle CalculatorRectangle(){
                int width = balance;    int height = 2 *
                   balance;
                Rectangle r = new Rectangle(width, height);
                return r;}
        //----------------------------------------
        public int[] TestArrays(){
                int[] x = {1,2,3};
                return x;       }
        //---------------------------------------
        public int[] Test2(){
                int[] x = {1,2,3};
                return x;       }
        //---------------------------------------
        public String reverseIt(String source) {
                int i, len = source.length();
                StringBuffer dest = new StringBuffer(len);
                for (i = (len - 1); i >= 0; i--){
                        dest.append(source.charAt(i));
                        }
                return dest.toString();
                }
        //--------------------------------------
        public String toString(){
                return Integer.toString(balance);
        }
}
```

Figure 5.1: Calculator.java

```
I am entering main
Percentage Similarity: 100
9 methods were found before filtering

4 Methods to be tested after filter
2 equivalence classes of Methods to be tested after filter
I am out of main
----------------------------------------
                  Cloneclass 0:
----------------------------------------
genfiles.Calculator.TestArrays Location: lines 23 - 25, Clone size:
   3 LOC

genfiles.Calculator.Test2 Location: lines 27 - 29, Clone size: 3 LOC
-----------------------------------------
                   Cloneclass 1:
-----------------------------------------
genfiles.Calculator.OutlineCalculator Location: lines 15 - 16, Clone
    size: 2 LOC

genfiles.Calculator.CalculatorRectangle Location: lines 18 - 21,
   Clone size: 4 LOC
------------------------------------------------
Number of Clones: 4
Number of Clone Classes: 2
Maximum Clone Size : 4 LOC
Maximum Clone Class Size : 2 clones
Minimum Clone Size: 2 LOC
Minimum Clone Class Size: 2 clones
Average Clone Size: 3.0 LOC
Average Clone Class Size: 2.0 clones

Length of program execution: 0.14 seconds
```

Figure 5.2: Sematic Method Clones found in *Calculator.java*

Since the semantic clones are deliberately added to the test code (except in the case of JSCTracker), the complete corpus of SMCs is known. This makes it easy to compute the precision and recall values. In the pilot test I am able to identify all of the SMCs, with 100% precision and recall—meaning that there are no false positives or false negatives. JSCTracker reflectively identifies 4 SMCs in the JSCTracker code. They were 2 pairs of static methods that had been duplicated in two different classes. These methods were also syntactic clones.

5.2 Case Study

Twelve relatively large samples of Java open-source software belonging to a variety of domains (shown in Table 5.1), are used to validate my tool and algorithm. Each sample project is checked for SMCs using JSCTracker and the SMC data analyzed.

Table 5.1: Open source Java code evaluated for SMCs

Software	**Domain**	**Size in LOC**
hsqldb-2.29	Java database engine	296,408
apache-commons-lang3-3.1	Standard Libraries	218,937
findbugs-2.0.1	Java based code analyzer	186,069
sablecc-4	Compiler generator	176,848
jabRef-2.6	Reference management	102,015
freemind-0.8.1	Mind-mapping	72,598
jetty-6.1.2.4	Web-services	50,516
jhotdraw-7.0.6	Framework for creating drawing editor	50,483
openJava-1.1	Extensible Java language	39,353
doctorj-5.0.0	Code Analyzer	38,399
quilt-0.6	Test coverage evaluator	9,654
importscrubber-1.4.3	Code beautifier	2,062

5.3 Results

The results of the case study are recorded for the number and size of SMCs and SMC classes identified. The total execution time of the analysis is also recorded. These results are then analyzed to answer the 4 research questions.

5.3.1 Clone Detection Results

Research Question 1: *Can method IOE-Behavior analysis be used in practice to detect SMCs in Java source code?*

Table 5.2 gives a summary of the results obtained from the analysis of the 12 code samples. The second column reports the number of actual SMCs found—actual clones refer to the detected clones, minus the false positives. Columns 3, 4 and 5 provide information on the maximum, minimum and average SMC size, in lines of code. The last three columns—7, 8, and 9 give the maximum, minimum and average SMC class size. The unit of measure is the clone. For example a SMC class with 3 clones has a size of 3.

Table 5.2: Analysis of detected clones

Software	**# Actual SMCs found**	**Max. SMC size in LOC**	**Min SMC size in LOC**	**Avg. SMC size in LOC**	**# SMC classes**	**Max SMC class size**	**Min SMC class size**	**Avg SMC class size**
hsqldb	6	6	2	3	2	4	2	3
apache	45	28	3	8	16	5	2	2
findbugs	19	22	3	2	6	4	2	1
jabRef	5	26	9	14	2	3	2	2
freemind	0	0	0	0	0	0	0	0
jetty	0	0	0	0	0	0	0	0
jhotdraw	2	4	4	4	1	2	2	2
openJava	8	3	3	3	4	2	2	2
doctorj	5	5	5	5	2	2	2	2
quilt	2	4	4	4	1	2	2	2
sablecc	15	4	4	4	6	5	2	2
importscrubber	0	0	0	0	0	0	0	0
Total	107							
Average	16.46	8.50	3.08	3.92	3.33	2.42	1.50	1.50

An average of 16 SMCs are detected in the 12 Java code samples tested. The actual numbers of detected SMCs ranges from 0 (in *freemind*, *jetty* and *importscrubber*), to 45 (in *apache*); with an average of approximately 5 false positives. The maximum number of SMCs are detected in *apache*, with almost 100% precision. Figure 5.3 on the following page shows the number of SMCs detected in each software sample—including false positives. The SMCs identified ranged in size from 2 LOC to 28 LOC. On average, the maximum detected SMC size is 8 LOC, while the average minimum is 3 LOC. From Figure 5.4 on the next page, the SMC sizes do not vary for the 8 smaller software samples. They have maximum and minimum SMC sizes that are equal to each other. The 4 largest software samples (*hsqldb, apache, findbugs, jabRef*) on the other hand, have SMC sizes ranging from 17–25 LOC in *apache, findbugs and jabRef*, to 4 LOC in *hsqldb*.

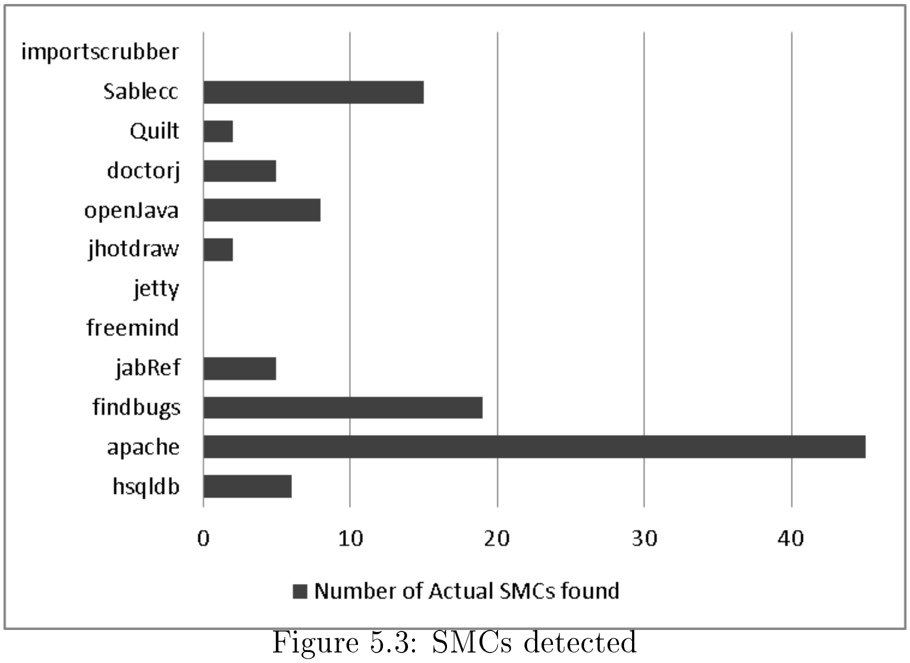


Figure 5.3: SMCs detected

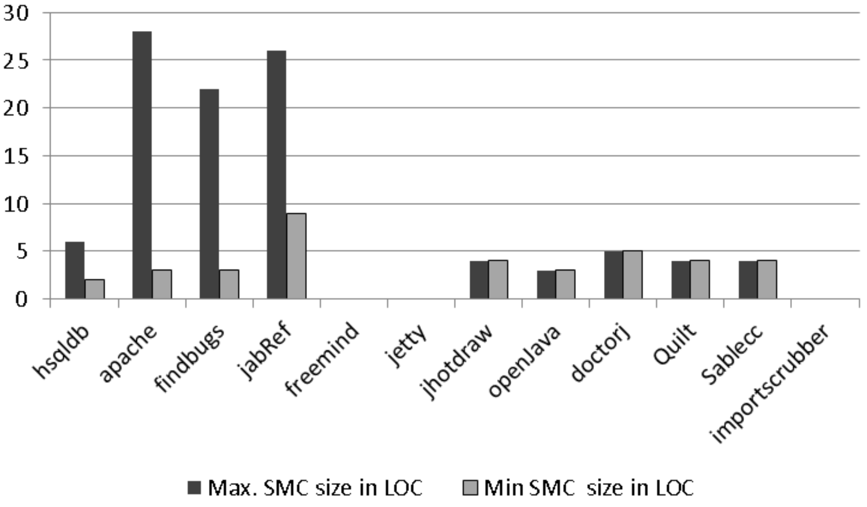


Figure 5.4: The maximum and minimum sizes SMCs detected

On average, the maximum number of SMC classes identified in any of the tested software is 3. The actual number of SMC equivalence classes identified range from 0 in *freemind*, *jetty* and *importscrubber* to 16 for *apache*. Between 1 and 6 SMC equivalence classes are detected in the other code samples.

Generally, the SMC class size is 2. The four largest samples of software have SMC classes with sizes 3–5. Of the smallest 8 samples of software, only *sablecc* has SMC class size greater than 2. A SMC equivalence class size of 5 is detected for the test software *sablecc*.

5.3.2 Efficiency of the Algorithm

Research Question 2: *Do the pre-testing filters used in method IOE-Behavior analysis improve the precision and efficiency, of the SMC detection process?*

Table 5.3: How pre-filters reduce methods requiring dynamic testing

Software	**# Methods before Effects filter**	**# Methods after Effects filter**	**% reduction**
hsqldb	2724	2126	22
apache	1320	867	34
findbugs	429	242	44
jabRef	1361	862	37
freemind	1416	863	39
jetty	998	746	25
jhotdraw	210	110	48
openJava	699	556	20
doctorj	1328	992	25
quilt	623	420	33
sablecc	1317	853	35
importscrubber	131	76	42
Total	12556	8713	404
Average	1046	726	33.67
Median	1158	800	34.77

The efficiency contribution of my algorithm over existing methods, is evaluated as the reduction in the number of methods that require dynamic testing. In the algorithms used by Jiang *et al* [2] and Deissenboeck *et al* [1], prior to testing, candidate clones are placed into equivalence classes based on input-output behavior. Table 5.3 on the preceding page shows the difference in the number of methods that require dynamic testing, using this type of algorithm and my proposed IOE-Behavior algorithm, for SMC detection. It shows that using IOE-Behavior with its effects filter, reduces the number of methods requiring testing by a minimum of 20% (in *openJava*) and a maximum of 48% (in *jhotdraw*). The average reduction is approximately 34%. Eight of the 12 samples of software analyzed have the average or greater than the average reduction in the number of methods to be tested. Recall from Section 3.5.2 on page 60, that a reduction in the number of methods, means a quadratic reduction in the number of tests run. Thus, for 100 methods, considering the minimum reduction recorded—20%, when 100 methods are reduced to 80, there is a significant reduction in the number of tests run by 20^2. The impact is even greater for higher percentages of method reduction.

5.3.3 Comparison of Results for IOE-Behavior and Input-output Analysis

Research Question 3: *How does SMC detection using method input-output behavior (obtained from methodType information), compare to that of SMC detection using method IOE-Behavior?*

To answer the third research question, the performance of JSCTracker for detecting SMC's using method IOE-Behavior is compared to similar detection, using method input-output only. The results for the latter are obtained by using the methodType analysis option in JSCTracker. In this analysis a method's return type and its parameter types are used to generate equivalence classes. Methods in each equivalence class are then run and the SMCs are identified as those methods that return the same value when given the same inputs. Void methods cannot be tested in this way, since they have no return value. In an over-approximation, all void methods of the same methodType, are returned as SMCs. Table 5.4 presents the results of SMC detection using IOE-Behavior versus input-output. Three major differences are evident in the results. Three striking differences are in the number of clones detected, the number of false positives and the execution time.

Table 5.4: SMC detection and analysis using method IOE-Behavior vs. input-output

Software	# **SMCs found using IOE-Behavior**	# **false positives with IOE-Behavior**	**Execution time in secs using IOE-Behavior**	# **SMCs found using only input-output**	# **false positives with input-output**	**Execution time in sec using only input-output**
hsqldb	10	4	45.183	1825	1817	26.981
apache	46	1	11.582	165	106	7.537
findbugs	24	5	4.524	108	87	3.042
jabRef	7	2	10.159	538	533	4.167
freemind	2	2	7.731	474	474	0.484
jetty	0	0	5.765	437	433	1.092
jhotdraw	2	0	2.309	26	24	1.904
openJava	8	0	1.317	84	64	0.358
doctorj	7	3	1.635	799	791	3.964
quilt	14	12	6.463	285	281	1.233
sablecc	45	30	2.795	647	627	0.796
import-scrubber	0	0	1.873	30	30	0.39
Total	165	59	96.66	5418.00	5267.00	24.97
Average	13.75	4.92	8.44	451.50	438.92	2.08

Using method IOE-Behavior, an average of approximately 14 SMCs are detected per project. The maximum number (46) is detected in *apache*, a close second is 45 SMCs detected in *sablecc*; while 0 SMCs are identified in *importscrubber* and *jetty.* Using only input-output behavior, an average of approximately 452 SMCs are detected. In half of the code samples the number of detected SMCs is above the average or slightly below. The maximum number (1825) of SMC's is detected in *hdsql*. The minimum number of SMCs (26) is detected in *jhotdraw*. The number of SMC's detected in 10 out of the 12 projects analyzed using method IOE-Behavior, are less than the minimum number of those identified using input-output behavior only.

The SMCs identified using input-output have a much higher incidence of false positives than SMC detection using IOE-Behavior. Figure 5.6 on the next page demonstrates the percentage of false positives for each of the 12 samples of test code when input-output or IOE-Behavior is used. For all but one of the code samples, the percentage of false positives is greater than 70% for the input-output analysis. The only exception *apache* reports a false positive rate of 64%. For 7 of the code samples (*hsqldb, jabRef, freemind, jetty, doctorj, quilt, sablecc* and *importscrubber*), the detected SMCs had over 95% false positives. This includes *importscrubber* and *freemind* with a false positive rate of 100%.

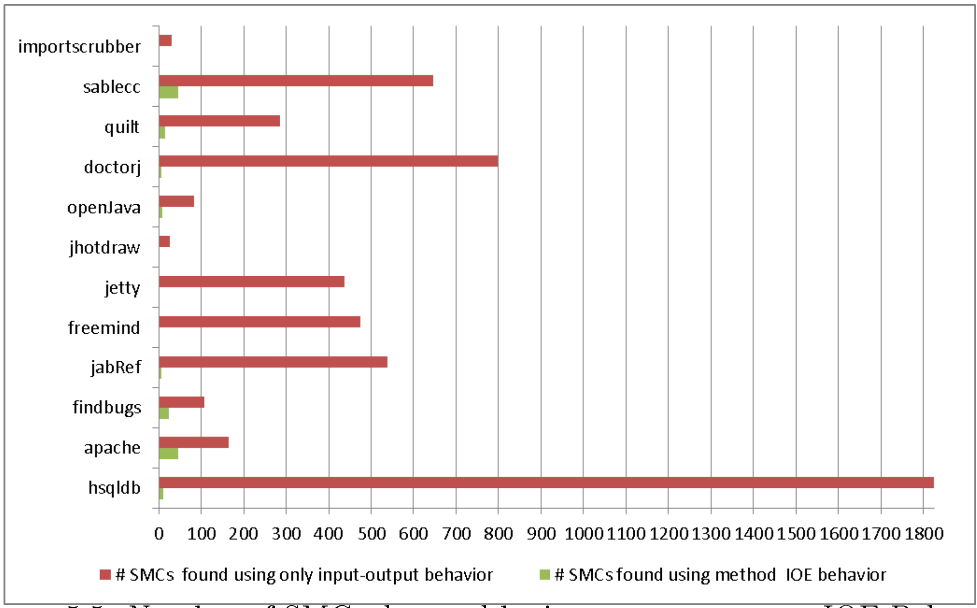


Figure 5.5: Number of SMCs detected by input-output versus IOE-Behavior

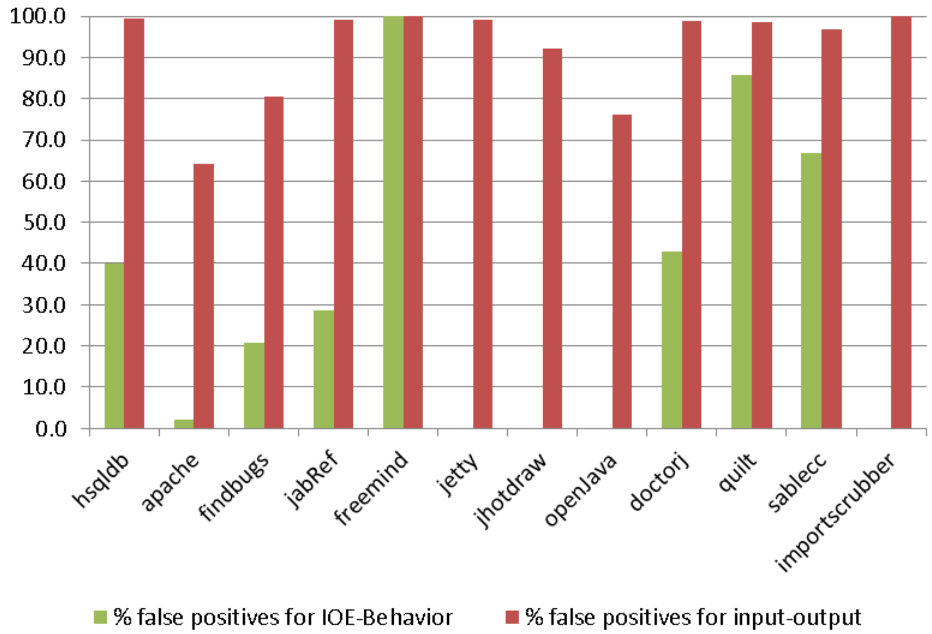


Figure 5.6: Percentage of false positive for IOE-Behavior analysis vs. input-output

The detection of SMCs using IOE-Behavior identified fewer false positives. For 9 out of the 12 code samples less than 42% of the identified SMC's are false. The 3 remaining code samples *freemind*, *quilt* and *sablecc* had 100%, 86% and 67% respectively.

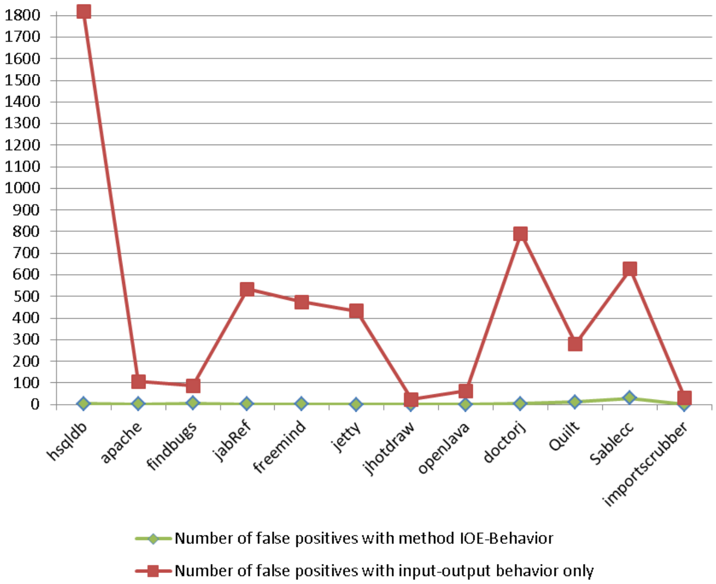


Figure 5.7: Number of false positives for IOE-Behavior versus input-output

Figure 5.7 on the preceding page shows the relationship between the actual number of false positives identified for each of the two types of analyses—input-output and method IOE-Behavior.

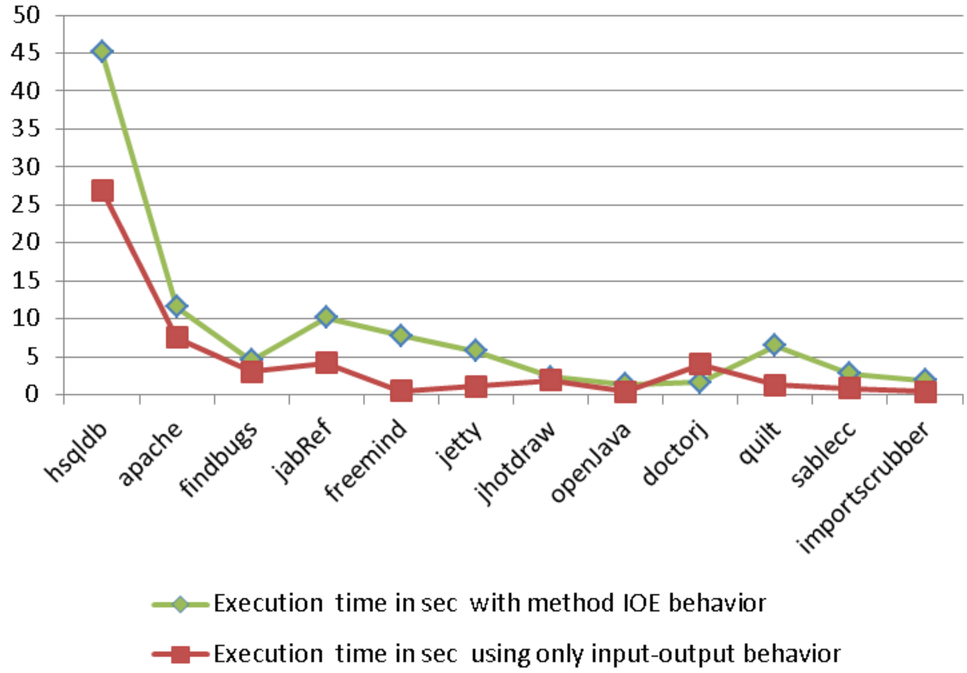


Figure 5.8: Execution time for analysis using method IOE-Behavior vs. input-output

Generally, the method IOE-Behavior analysis had execution times on average over 4 times slower than input-output analysis—8.44 seconds versus 2.08 seconds. The actual differences in time ranges from over 7 times slower for *freemind*, approximately 2 times slower for *hdsql*, *apache* and *jabref*; while less than 1 second slower for *jhotdraw*. The one outlier in this data is *doctorj*, for which the execution time is almost 2 times faster for the method IOE-Behavior analysis than the input-output analysis. One possible explanation for this is that the latter analysis involved the processing of almost 800 detected SMCs (791), while the former returned a mere 7 clones. Figure 5.8 shows the actual execution times for the two types of analysis; while the relative speed of the two types of analyses, (expressed as speed-up of input-output analysis) is shown in Figure 5.9 on the following page. From

Figure 5.9 it is evident that as the code size of the software samples decreased, the relative execution time speedup increased. Two outliers to this observation are *freemind* and *doctorj*. For both of these code samples, the execution time is smaller than that of the two smallest code samples.

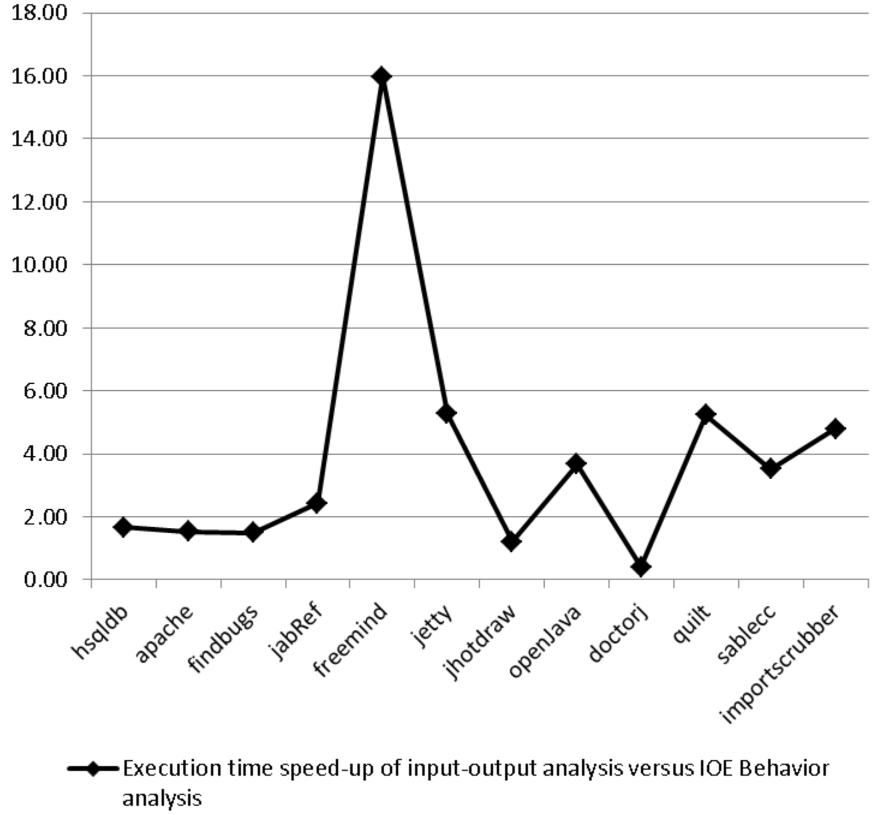


Figure 5.9: Execution time speed-up of input-output analysis versus IOE-Behavior

Deissenboeck *et al* [1] use an input-output type of analysis to detect semantic clones. Table 5.5 on the next page shows the number of SMCs identified by JSCTracker and by Deissenboeck *et al* for the same software. In all cases, except for *jabRef* and *apache*, JSCTracker detects less clones than reported by Deissenboeck *et al*. Although I use the same version of the software as in Deissenboeck *et al*, the number of clones identified in the two studies, cannot be strictly compared, since theirs does not provide precision and recall values. However, considering the difference in the number of clones reported when using the input-output method, compared to the IOE-Behavior analysis, it suggests that there may be false positives in the clones reported by Deissenboeck *et al.* It should be noted also, that 7 of the source code samples used in my research, are not analyzed by Deissenboeck *et al.*

Table 5.5: Comparison of JSCTracker Results to Deissenboeck *et al* [1]

Software	**# Actual SMC found by JSCTracker**	**# SMC detected in Deissenboeck *et al***
apache	45	54
jabRef	5	55
freemind	0	11
jetty	0	15
jhotdraw	2	18

5.3.4 Reliability of JSCTracker and IOE-Behavior Analysis

Research Question 4: *How reliable is method IOE-Behavior for the detection of SMCs?*

The reliability of my method IOE-Behavior analysis for SMC detection is determined using precision and recall measures as defined in Section 2.1 on page 7.

Table 5.6 gives the execution times and a summary of the accuracy of the results obtained from the analysis of the 12 code samples. The second column of the table shows the *SMC corpus*— the total number of SMCs found in the software. This is obtained in a two part process. First, an over approximation of the clone corpus is obtained by running JSCTracker using only the methodType filter. This set is further refined by manually verifying the returned clones. The methodType filter method, returns a large number of clones (up to 1825), because of the size of the test projects and also because all void methods of the same type are returned as SMC candidates without further testing. The same is also true for abstract methods. Hence, for practical reasons, only a randomly selected sample of the identified SMC classes are verified manually. For results with less than 10 equivalence classes, all of the classes are manually verified. For results with greater than 10 equivalence classes, 10 or 50% (whichever is greater), of the classes are selected at random for manual verification. The third column gives the number of SMCs detected by JSCTracker. The next three columns provide information about the accuracy of the clone detection process. The false positives are the number of incorrect clones reported, while the false negative column holds the number of clones that exist in the corpus, but are not identified by the tool. The precision column shows the degree of correctness of the detected SMCs, while the recall column presents the percentage of the corpus that is detected. The Execution time of the clone detection process is shown in the last column.

Table 5.6: Summary of Analysis Results

Software	**Size of SMC corpus**	**# SMCs found**	**# False Positives**	**% Precision**	**# False Negatives**	**% Recall**	**Time in seconds**
hsqldb	8	10	4	60	2	75	45.183
apache	59	46	1	98	14	76	11.582
findbugs	21	24	5	79	2	90	4.524
jabRef	5	7	2	71	0	100	10.159
freemind	0	2	2	0	0	na	7.731
jetty	4	0	0	na	4	0	5.765
jhotdraw	2	2	0	100	0	100	2.309
openJava	20	8	0	100	12	40	1.317
doctorj	8	7	3	57	4	50	1.635
quilt	4	14	12	14	2	50	6.463
sablecc	15	45	30	33	0	100	2.795
import-scrubber	0	0	0	na	0	na	1.873
Total	146	165	59	613.18	40	681.75	96.66
Average	12.17	13.75	4.92	68.13	4	75.75	8.06

The computation of precision is done through automated testing. Each returned SMC class is resubmitted to JSCtracker and tested with a large number of tests (30^2 to 100^2

compared to the 20^2 used in the standard detection). Starting with 30^2 tests, and increasing in 10^2 increments, the class is tested as described in Section 3.5 on page 52, until the clone classes returned, reach a fixed point. Because of the limitations of class size in Java, in some test cases, 70^2 tests are not possible. In such cases, the verification of precision consists of a combination of automated tests and manual checks of the SMC class.

The recall analysis is completely manual. The *SMC corpus* or body of clones found in any code sample, is obtained in a two part process. First, an over approximation of the SMC corpus is obtained by running JSCTracker using only the methodType filter. This set is further refined by manually verifying the returned SMC equivalence classes. Because a large number of clones may be identified, to keep the process practical, only a randomly selected sample of the clone classes are verified manually. For results with less than 20 equivalence classes, all of the classes are manually verified. For results with greater than 20 equivalent classes, 50% of the classes are selected at random for manual verification. The recall value for any analysis is then computed as the percentage of the SMC corpus that is identified.

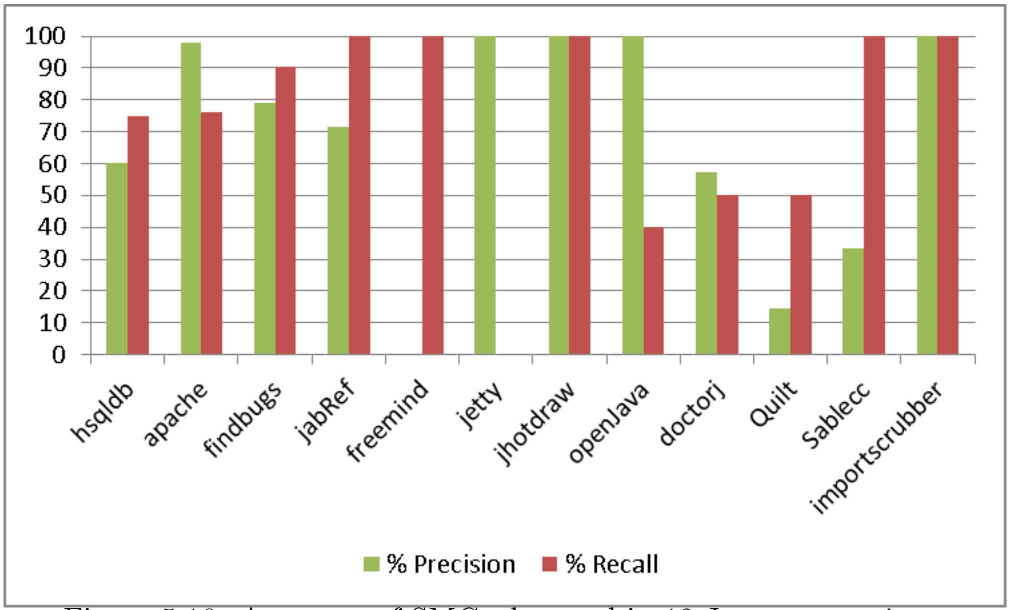


Figure 5.10: Accuracy of SMCs detected in 12 Java test projects

This is described in detail in Section 2.1 on page 7. The use of the results of the methodType filter to obtain the SMC corpus is valid, since any equivalence class of clones returned by this filter, has the same return type and parameter types. It is thus a valid super set of the set of clones with the same IOE-Behavior.

From the accuracy analysis, the average precision value is 68%. Thus on average, approximately 7 out of 10 SMCs identified, are actually clones. The average recall value is about 76%. This means that on average, over 3 out of every 4 clones in the identified SMC corpus are detected as SMCs by my algorithm and the tool. SMCs are detected with 100% precision in 2 of the tested projects—*jhotdraw* and *openjava*, while 98% precision is obtained for *apache*.

CHAPTER 6
DISCUSSION

This chapter is divided into five sections. The first explains how the assumptions of this study impact the results obtained. The second discusses design decisions, explaining why they are made. Both of these sections also provide suggestions of alternate approaches that may improve results and should therefore be explored. The third discusses the results in the evaluation chapter. This is followed by a discussion of the limitations of the current work. The chapter ends with suggestions for future work.

6.1 How Assumptions Impacted Case Study Results

This section discusses assumptions used to make design decisions or to define the scope of my research.

One of the key contributions of this research is the inclusion of a method's effects in determining its semantics. I devised the static non-local variable mutation (NLVM) analysis, described in Section 2.3.2.2 on page 18 to determine a method's effects. In this analysis, the effects of a method are given as the set of non-local variables that it mutates. This includes instance variables, class variables and object parameters (and recursively the instance and class fields of those objects). The assumption is that these are the only components of the heap that a method can alter. However, as a result of this assumption the effect of a method on the system clock and system output streams are not considered in the NLVM effects analysis. Consequently methods that output different strings to the console are included in the definition of SMC although they should not be. Alternatively, persistent changes to system variables such as the output stream and JVM states could be tracked separately and included in the NLVM analysis of effects.

One of the assumptions of my research as described in Section 3.1 on page 39, is that all classes have at least one constructor which sets its instance variables. This assumption is true for most classes. However, there are classes for which it is not so. When random objects of these classes are generated, the returned object is always in the same state—the instance variables are all set to the Java default values for their type. This reduces the number of input states tested, since only one state of the object is being generated. The result is a higher occurrence of false positives, where a difference in an object's structural state, would produce different results. For example, considering a class *A* with instance variables **int** *val* and **boolean** *state*, if *A* has no constructor that sets its instance field values, whenever an object of *A* is created by the *randomObjectCreationString* method shown in Figure 3.11 on page 55, the instance variables *val* and *state* will always have the Java default **int** and **boolean** values of *0* and *false* respectively. Thus, the methods *setState* and *resetState* as shown in Figure 6.1 on the following page, will be falsely identified as semantic method clones—SMCs. This is because *a.val* will always be *0* and thus *isEven* will be true. Hence *a.state* is always set to true, and the two methods which have the same effects, will also appear to always return the same heap state. This constructor limitation has less impact for methods that take multiple parameters, some of which are not affected by the constructor problem; since the other parameters can contribute to creating differences in input state.

One approach to eliminating this problem, would be to check the side effects of a selected constructor. If its effects do not include the instance variables of the class, then the relevant methods would be called to initialize the instance variables.

```
private void setState(){
   if(a.val == 0)
     a.state = true;
}
```

```
private void resetState(){
   if(a.val.isEven()){
     a.state = true;
   }
   else{
     a.state = false;
   }
}
```

Figure 6.1: Methods of class *A* that can lead to false positives

In the dynamic testing phase of my algorithm, the results or outputs of methods are converted into a *String* for comparison purposes to detect semantic equivalence. When methods return an object, it is assumed that the object returned has a *toString* method that prints out more than just the class name or its memory address. It is assumed that the string returned distinguishes the object's state by including information derived from instance variable field values. An alternative to this assumption would be to use AspectJ to override the *toString* method to ensure that it prints out the values of the structural components of the object. I tested the usefulness of such a method using the *newToString* method whose pseudocode is shown in Figure 6.2 on the next page. This method prints out the state of an object as a table of field names and their corresponding values. For non-primitive values, the method recursively prints the field values. It includes a time out feature to prevent infinite loops for circular objects. Adding this method to the implementation of JSCTracker greatly increased the execution time to analyze projects by over 2 minutes. As a result, I decided that the advantage of using this method (guaranteeing the output of the *toString* method), is outweighed by its disadvantage (namely the increase in execution time). Thus this method is not incorporated into the tool.

```
Final int timeout=12;
pointcut overrideToString(Object a): call(* String Object.toString()) && target(a) && (withincode(private EffectRec utility.MethodTests+.pRun*())||
withincode(private EffectRec utility.MethodTests+.stateMethod*()) ||
withincode(private EffectRec[] utility.MethodTests+.pMethod*()));
//----------------------------------
String around(Object a): overrideToString(a){
  try{String str  = newToString(a,0);
    return str;
  }
  catch(Exception e){return null;}
}
//----------------------------------
private String newToString(Object o, String status, int depth){
  if(depth == timeout)return status;
  else{
    depth++;
    StringBuilder sb = new StringBuilder();
    sb.append(status);
    for each field i of o{
      if(field[i].isPrimitive()){
        sb.append(field[i].name + ":␣" + field[i].toString());
      }
      else{
        sb.append(field[i].name + ":␣");
        sb.append(newToString(field[i],sb.toString(),depth));
      sb.append("\n");
      }
  }
 }
}
```

Figure 6.2: Pseudo code for *newToString*

6.2 Design Considerations

Denotational semantics and symbolic execution are two alternatives that I explored for checking for semantic equivalence between methods. However, I chose not to use either, because finding a unique, normal form expression or equation to represent the semantics of a method is undecidable and a good approximation difficult to achieve. Thus these approaches do not help check for semantic equivalence. Consequently, I choose to use a method's observable behavior: the combination of input-output and effects which I refer to as a method's IOE-Behavior. I selected this option since it is an easily computable approximation for this undecidable problem.

The SMC detection algorithm is language independent. However, the implementation for testing is done in Java. Java is selected because JSCTracker is built around the JastAdd compiler generator (described in Section 2.5 on page 30); and at the time of development this is the only language supported by JastAdd. Java is also a good language choice because of its portability, the availability of extensive libraries and Integrated Development Environment (IDE) support like Eclipse. However, there are some cons to using Java especially with respect to space usage.

A particular problem for space usage is HashMaps. These data structures (HashMaps), are used at several points in the implementation of my algorithm. This type of data structure is selected because it is time-efficient. For example, a HashMap is used to store methods on the key of their methodType and effect. This is a quick way to generate equivalence classes post filtering. A HashMap is also used to store the results of methods during the testing stage. Since the result is used as the HashMap key, methods with the same result hash to the same bucket. This eliminates the need for comparing each method to every other method in the equivalence class, to determine semantic equality. However, while the use of a HashMap renders the algorithm more efficient in some ways, because the implementation is in Java, this also causes the JSCTracker application to be memory intensive. This can be optimized though, if code can be written to manipulate or control garbage collection; allowing for HashMaps to be disposed of once they are no longer needed. However, since Java handles its own garbage collection, the application is not always as efficient as it might be, if it were implemented in C or C++, where the developer has control over memory allocation and deallocation.

In the first iteration of JSCTracker, private methods were not analyzed for semantic equivalence. The logic was that these methods should not be visible to the analyzer, since this would violate Java's visibility rules. I later decided to include such methods as they are part of real world projects and should be analyzed for in-house code maintenance. Privileged aspects described in Section 2.7 on page 34 are used to provide this facility. The use of AspectJ also allowed for bypassing calls to the Java *Object.equals* method, replacing it with a method that evaluates structural equivalence—*newEquals* shown in Figure 3.22 on page 65.

There are some problems with using AspectJ though. The weaving of aspects necessary for running the application, is time-consuming. There are also issues with AspectJ with regards to limitations on aspect size. I had to take care in the automatic file generation process to ensure that the generated aspect does not exceed the maximum size allowed by the JVM.

Another possible drawback of using AspectJ is that there are dependencies in the code of my JSCTracker tool that may not be obvious. For example modification of the code can possibly change how pointcuts are handled. This can alter the behavior of the application, since an advice may then be executed at points where it is not intended; or it may not be executed at program points where it is required. For example, in the *AMT1Testcase* pointcut in Figure 6.3, the matching joinpoint (shown in line 1 of the code), is a call to a constructor of the *AMT1* class, which takes no parameters. The pointcut provides an advice (shown in lines 2–4) that runs the *RunTest* method after completing the constructor call.

```
1 pointcut AMT1Testcase(): call(AMT1.new());
2 after() returning(AMT1 a): AMT1Testcase(){
3 a.RunTest();
4 }
```

Figure 6.3: Sample JSCTracker pointcut

The identification of this pointcut and execution of the related advice can be affected if any of the following occur:

- the signature of the constructor is changed to include parameters
- the name of the test class is changed from *AMT1*

If either change occurs, the pointcut *AMT1Testcase* is not identified, so the advice will not be invoked, thus the method *RunTest* will not be executed. Hence, if any of these changes are made to the code, it should be reflected in a matching modified definition of the pointcut or the creation of a new pointcut; otherwise the advice will not work as expected. This problem is only possible, for persons who are allowed maintenance access to the JSCTracker code, since regular users of the tool cannot access its code. Providing proper documentation for the tool can reduce the probability of such problems.

When comparing methods for equivalence, I decided not to consider sub-typing in determining methodTypes or determining equivalence of effects. For example considering two classes *A* and *B*, such that *B* is a subclass of *A*. The two methods MethA1 and MethB1 with the following signatures:

$$int\ \ MethA1(int\ \ x, A\ \ o1)$$
$$int\ \ MethB1(int\ \ x, B\ \ o1)$$

will not be stored in the same equivalence class and thus never checked for equivalence.

I made this decision since the subtype relationship is not symmetric and thus would not produce equivalence classes as required by the algorithm. Also, Java's type checking rules do not allow for replacing one method with another that has different parameter types in all cases, hence, such methods would not be both able to replace each other.

6.3 The Case Study Results

Two marked patterns of the case study results are the high number of false positives for the input-output SMC analysis (as used in related work) and the difference in execution time between the input-output SMC detection and method IOE-Behavior analysis (Section 5.3.3 on page 88).

An explanation for the high number of false positives in algorithms that do not consider method effects is that such an input-output analysis does not use any dynamic testing to determine semantic equivalence of void methods. Instead, all void methods of the same type are assumed to be SMCs. It is this failure to test these methods that results in the low precision values (high number of false positives), for the input-output analysis of SMC. Figure 6.4 on the next page compares the number of void methods found in the 12 test projects against the number of false positive SMCs identified. The two lines in the line-graphs are the same general shape, indicating a relationship between the number of false positives and the incidence of void methods in the code. For *hsqldb* and *doctorj* the number of false positives exceeds the number of void methods indicating that there are false positives which are not void methods. For *jabRef*, *freemind* and *jetty* the number of void methods exceeds the number of false positives indicating that some of the void methods are actual SMCs.

An alternate approach for the input-output analysis would be to exclude all void methods from analysis. This would improve *precision*, by reducing the number of false positives. However, it would reduce *recall* since the number of false negatives would increase, as void SMCs would go undetected. The issues created by void methods, support the use of a method's effects in determining semantic equivalence.

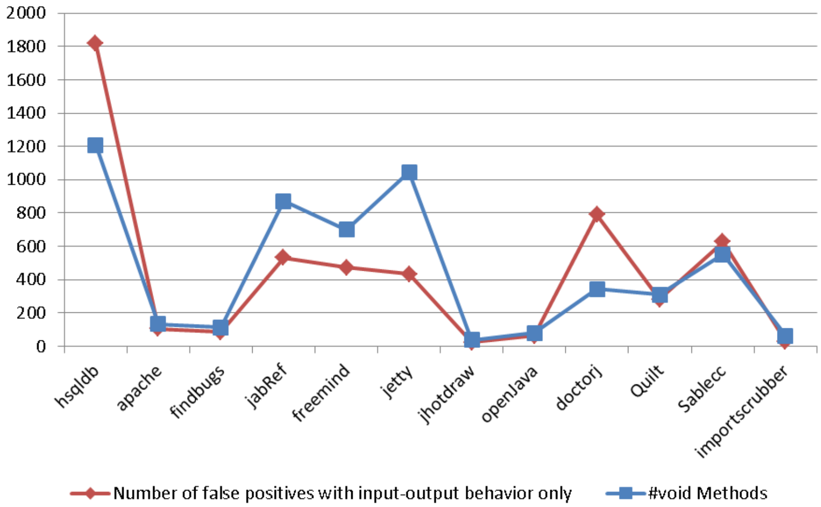


Figure 6.4: Percent void methods vs. percent false positives for input-output

The second prominent result of the case study, is the execution time difference between analysis using method IOE-Behavior versus that using input-output behavior. To investigate this trend further, I removed the void methods from the set of SMCs returned by the input-output analysis. Further analysis of the execution times, by computing the average time taken for each method, shows that, when ignoring void methods, on average method IOE-Behavior analysis of SMCs takes 0.010 seconds per method while input-output analysis takes 0.015 seconds per method. A possible explanation is that the IOE-Behavior analysis is faster since the extra filter used in this method reduces the number of methods (in each equivalence class) actually tested. Figure 6.5 on the following page shows the relative execution times per method for each of the code samples. There is one outlier—*openJava* for which the input-output took considerably more time than the method IOE-Behavior analysis.

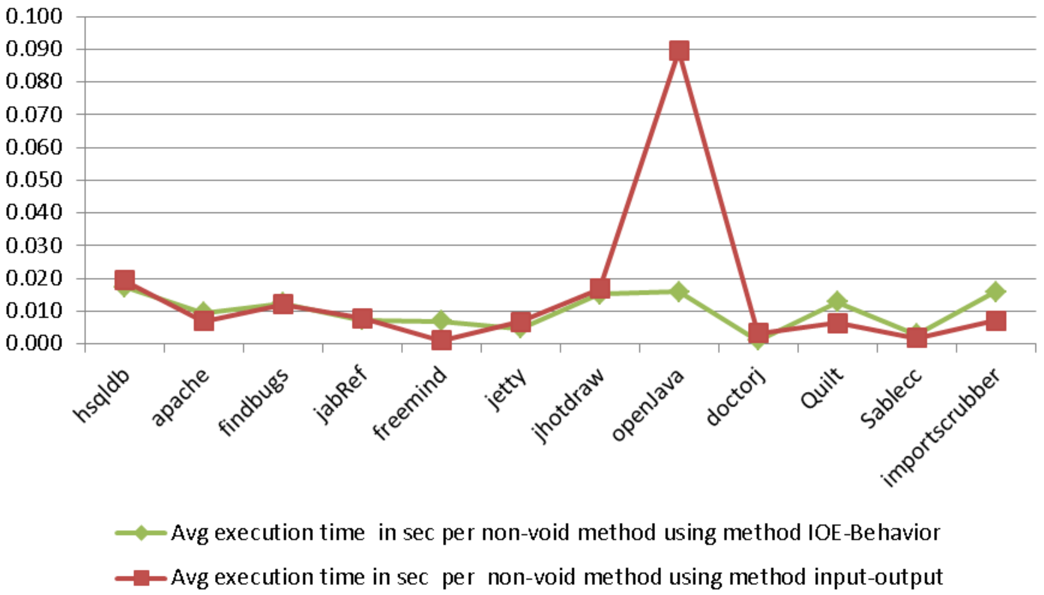


Figure 6.5: Average execution time per method for IOE-Behavior vs. input-output

6.4 Limitations

One of the primary limitations is the use of only an approximation of the input space of the tested methods. Since the input space is possibly infinite in size, only a small fraction of it can be tested. To make this sample representative of the whole, samples are generated at random using an unbiased process to ensure that all cases are equally likely. However, because all cases are not tested, the results will not be 100% accurate. In some cases, the randomly generated objects are not appropriate, particularly when specially formated data is required. For example with methods that require a file name or XML formatted string, my algorithm would generate a string, but the formatting test will fail and it would not be possible to run such methods for further testing. This issue can be addressed in future work by taking into account preconditions in the random generation of objects.

The computation of the accuracy of the algorithm is also an approximation. Each project contains thousands of lines of code and methods, so it is not practical to manually check each method to obtain the SMC corpus. Instead, a sample size of 50% of all equivalence classes are tested. Since all of the methods are not tested, there is a possibility that the corpus is incomplete. Testing 50% provides odds of 1 in 2 that a method would be checked. This is a higher percentage (in some cases as high as 49% higher), than that used by Jiang and Su [2] in related work.

The use of filters is intended to improve efficiency. However, they may present some possible threats to recall. First, when using methodTypes, differences between types, such as *Double* and *double*, which might not make any functional difference in some abstract sense, would prevent the algorithm from grouping these methods together. There is a similar issue with subtyping. For example considering two methods *foo1* and *foo2* with an object parameter. If *foo1* takes a parameter of object type A and *foo2* takes an object of type B (where A is the supertype of B), our first filter will separate *foo1* and *foo2*. However, if the methods do not change any fields of the parameter, (or only change super class fields), they can possibly have the same observable behavior. But *foo2* cannot take arguments of type A. Second, since the NLVM effects analysis is conservative, there is the possibility that methods are split into different groups that might not really differ in their write effects. However, mere testing will not improve recall, since it only splits apart method groups, it does not make them form.

The assumption that the *toString* method is appropriate for all classes may not always be true. When this method is not overridden in a class, a call is made to the *Object.toString* method. Since $Object's$ *toString* method only returns a string containing a class name and an address, when the string is used in structural equivalence comparisons, it can lead to false negatives.

The evaluation of a method's effects include changes to its receiver object if any, plus changes to non-local variables. This is not a complete list of all of the possible effects of a method. Changes to the system variables and external devices, such as file outputs also need to be considered.

6.5 Future Work

For future work I would like to implement JSCTracker for C++ or C#. This would also mean using a different compiler generator which can handle either of these two languages. I would also like to apply JSCTracker to grading students' programming assignments to provide qualitative feedback. This would be done by using the concept of degree of semantic similarity and an instructor-provided oracle. JSCTracker can also be used to identify refactoring opportunities in code. This can then be used as input for refactoring tools.

CHAPTER 7
CONCLUSIONS

This chapter summarizes the research problem addressed in my dissertation and the approaches applied to arriving at a solution. It also outlines the experimental results and provides a description of the contributions made to the research area.

7.1 Summary of Research

The determination of semantic equivalence between two code fragments is a formally undecidable problem. While the problem is undecidable, the thesis of my dissertation is that a good approximation can be reached, by using a combination of input-output and effects behavior of a method. I refer to this as a method's IOE-Behavior. In related work [1, 2], semantic equality between code fragments has been estimated by using automatically generated dynamic tests, to run methods. This dynamic testing involves generating parameter values for a method and tracking the results when that method is called with these parameters. The IOE-Behavior analysis also uses this approach, but in addition, it incorporates a new dimension: the method's effects. A method's effects refer to how its execution changes the heap. In my research, a method's effect information is used in two ways. First, it is used as a pre-filter to group methods of the same type (already filtered by method type), into equivalence classes based on effects. Equivalence classes of size 1 are omitted from further analysis. Hence, the use of the pre-filter eliminates unnecessary testing by removing unlikely clone candidates, before the testing phase. As a result, the overall number of methods that need to be tested, using the dynamic tests, is reduced. Secondly, a method's effects are considered as part of its output behavior, along with the return value. Thus, when using method IOE-Behavior, even void methods have output behavior that can be checked.

7.2 Overview of Results

My semantic method clone (SMC) detection algorithm using method IOE-Behavior, and my tool JSCTracker used to implement it, are validated using a case study with 12 large samples of Java open source code. The test projects are for applications in different domains (including database management systems and code analyzers); and range in size from approximately 2 KLOC to 297 KLOC— with an average of over 103 KLOC. The goal of the case study was to test the hypothesis that: *a good approximation to detecting semantic similarity between Java methods can be reached, by using method IOE-Behavior.* For investigation purposes, the hypothesis is broken down and expressed as 4 research questions.

The first research question is: "Can method IOE-Behavior analysis be used in practice to detect SMCs in Java source code?" This was answered in the affirmative as a total of 107 actual SMCs were detected in the test projects. The number of SMCs identified in individual projects ranged from 0 to 46; with an average of 16 SMCs. The number of SMCs detected increased with project size, with one exception. This suggests that the type of project may also impact the number of SMCs detected.

The second research question is: "Do the pre-testing filters used in method IOE-Behavior analysis improve the precision and efficiency, of the SMC detection process?" The results of the case study show that filtering the methods by type and then with effects information, reduces the number of methods to be tested dynamically, by an average of about 34%—a maximum of 48% and a minimum of 20%. This reduction in the number of methods could mean a quadratic reduction in the number of dynamic tests run. For example, considering a project with 100 methods, a 34% reduction in the number of methods, means that the number of tests required could be reduced by 34^2 tests. The elimination of the unnecessary tests increases the overall efficiency of the SMC detection algorithm.

The third research question is: "How does SMC detection using method input-output behavior (obtained from methodType information), compare to that of SMC detection using method IOE-Behavior?" To find an answer to this research question, the case study compares SMC detection using IOE-Behavior to the detection algorithms used in related work, which use only method input-output. The comparison is made on two levels—precision of results and execution time. The method IOE-Behavior analysis is more accurate, since void methods are not ignored or accepted as SMCs as a default. This is further supported by the case study results which show that the ratio of false positives returned when method IOE-Behavior is used, is on average 32% while it is 92% when only input-output (methodType), is used. The execution time seems better for the input-output algorithm, which has execution times ranging from less than half of a second to almost 27 seconds, while IOE-Behavior had execution times from less than 2 seconds to about 45 seconds. However, the input-output approach never tests the void methods, which make up on average 11% to 87% of the methods in the projects tested. A more accurate comparison of the execution times, therefore, is the average time per method. When the average execution time per Java method is evaluated, the times for input-output and method IOE-Behavior are only thousandths of a second different. Indeed on average, execution time per method for IOE-Behavior analysis is 0.005 seconds faster.

The forth research question is: “How reliable is method IOE-Behavior for the detection of SMCs?” To answer this question, the reliability of the algorithm and tool were assessed in terms of precision and recall of the SMCs identified. Precision and recall values of 68% and 76% respectively were reported by the case study results. This is a definite improvement over the 31% reported by Kawrykow and Robillard [4] but the same as that reported by Jiang *et al* [2]. The lowest recall value for a test project is 30%, while the highest is 100% reported for 25% of the projects tested. The average recall is 76%.

7.3 Contributions

The primary contribution of this research is the inclusion of the effects of a method in analysis of its semantic behavior. In the related work, researchers have begun to investigate semantic similarity between code fragments. They have employed different approaches, but at the heart of most of these is the use of dynamic testing [1, 2]. In this study, I present a novel approach to semantic method clone detection, which combines static and dynamic analysis in the detection of SMCs.

In previous research, only input-output behavior has been used in determining semantic equivalence. In such schemes the semantic equivalence of code fragments when run on the same input, is determined by the value that is output. This works for non-void methods. However, it does not work for void methods, since they have no return value to be used for comparison. Using method IOE-Behavior in semantic clone detection, allows for the evaluation of both void and non-void methods, since, the effects of a method are included in the comparison. Thus, even void methods have effects that can be used for comparison in determining semantic equivalence.

7.4 Conclusion

The motivation for my research is to detect semantic method clones, in Java software, using method IOE-Behavior and to evaluate the merits of my algorithm as it compares to existing related work. Both of these goals have been met as is evidenced by the results for the 4 research questions posed. It is my conclusion therefore, that method IOE-Behavior can be used to detect semantic method clones in Java software, with a reasonable degree of reliability. This is supported by the precision and recall values when compared to existing related work. The precision value of 68% is an improvement of 37% compared to that reported by Kawrykow and Robillard [4] but the same as that reported by Jiang and Su [2]. However, when comparing the heuristics used to measure precision, my method for measuring precision is more representational of the entire case study than Jiang and Su. The recall value of 76% is a 30% improvement over the only recall value that I could find, reported by Bellon *et al* [5] for the detection of syntactic clones.

APPENDIX
PILOT TEST FILES

1 HDelightCode.java File

```
package genfiles;



public class HDelightCode {
 int numb;
 public HDelightCode(){
   numb = 0;
 }

public static int flp2(int x) {
//returns the greatest power of 2 less than or equal to x
   x = x | (x >>> 1);
   x = x | (x >>> 2);
   x = x | (x >>> 4);
   x = x | (x >>> 8);
   x = x | (x >>>16);
   return (x - (x >>> 1))& 0xff;
}
//————————————————————————————————
public static int HighestPowerof2(int x){
//returns the greatest power of 2 less than or equal to x
   int tmp = x;
   int answer = 1;
   while(tmp > 1){
        answer = 2 * answer;
        tmp = tmp/2;
   }
   return answer;
}
//------------------------------------------------
public static int isqrt1(int x) {
//this method finds the nearest square root of x
  int x1;
   int s, g0, g1;

   if (x <= 1) return x;
   s = 1;
   x1 = x - 1;
   if (x1 > 65535) {s = s + 8; x1 = x1 >>> 16;}
   if (x1 > 255)   {s = s + 4; x1 = x1 >>> 8;}
   if (x1 > 15)    {s = s + 2; x1 = x1 >>> 4;}
   if (x1 > 3)     {s = s + 1;}

   g0 = 1 << s;                // g0 = 2**s.
   g1 = (g0 + (x >>> s)) >>> 1;  // g1 = (g0 + x/g0)/2.

   while (g1 < g0) {           // Do while approximations
          g0 = g1;                 // strictly decrease.
        g1 = (g0 + (x/g0)) >>> 1;
   }
   return g0;
}
//------------------------------------------------------------
public static int findSquareRoot(int myNumber){
 //find the nearest square root of myNumber
 final double EPSILON = .00001;
 int  guess = 1;
 double root = Math.sqrt(myNumber);
        while (EPSILON < Math.abs(Math.pow(root, 2) - myNumber))
        {
         guess++;
        }
        return (int)root;
  }
//------------------------------------------------------------
// Reversing bits in a word, basic interchange scheme.
 public static int rev1(int x) {
        x = (x & 0x55555555) <<  1 | (x & 0xAAAAAAAA) >>>  1;
        x = (x & 0x33333333) <<  2 | (x & 0xCCCCCCCC) >>>  2;
        x = (x & 0x0F0F0F0F) <<  4 | (x & 0xF0F0F0F0) >>>  4;
        x = (x & 0x00FF00FF) <<  8 | (x & 0xFF00FF00) >>>  8;
        x = (x & 0x0000FFFF) << 16 | (x & 0xFFFF0000) >>> 16;
        return x;
 }
//————————————————————————————————
public static int rev14(int x) {
 //reverses the bits in a word
   x = shlr(x, 15);              // Rotate left 15.
// x = (x << 15) | (x >> 17);    // Alternative.
   x = (x & 0x003F801F) << 10 | (x & 0x01C003E0) |
        (x >>> 10) & 0x003F801F;
   x = (x & 0x0E038421) <<  4 | (x & 0x11C439CE) |
        (x >>>  4) & 0x0E038421;
   x = (x & 0x22488842) <<  2 | (x & 0x549556B5) |
        (x >>>  2) & 0x22488842;
   return x;
}
//—————————————————————————————
public static int shlr(int x, int n) {
   return (x << n) | (x >> (32 - n));
}
//—————————————————————————————
public static int ffstr11(int x, int n) {
//find first string of n 1's in number x
   int k, p;
   p = 0;                // Initialize position to return.
   while (x != 0) {
    k = nlz(x);        // Skip over initial 0's
```

```
        x = x << k;       // (if any).
        p = p + k;
        k = nlz(~x);      // Count first/next group of 1's.
        if (k >= n)       // If enough,
           return p;      // return.
        x = x << k;       // Not enough 1's, skip over
        p = p + k;        // them.
  }
  return 32;
}
//————————————————————————————————————————————————
public static int ffstr12( int x, int n) {
//find first string of n 1's in number x
   int s;
   while (n > 1) {
        s = n >>> 1;
    x = x & (x << s);
        n = n - s;
   }
   return nlz(x);
}
//——————————————————————————————————————————————————
public static int nlz(int x) {
//returns the number of leading zeros in a word
        int n;
```

```
          if (x == 0){
           return(32);
          }
          n = 0;
          if (x <= 0x0000FFFF) {
           n = n +16; x = x <<16;
          }
          if (x <= 0x00FFFFFF) {
           n = n + 8; x = x << 8;
          }
          if (x <= 0x0FFFFFFF) {
           n = n + 4; x = x << 4;
          }
          if (x <= 0x3FFFFFFF) {
           n = n + 2; x = x << 2;
          }
          if (x <= 0x7FFFFFFF) {
           n = n + 1;
          }
          return n;
}
//------------------------------------------------------------
public static int nlz2a(int x) {
//returns the number of leading zeros in a word
    int y;
    int n, c;
    n = 32;
    c = 16;
    do {
          y = x >>> c;  if (y != 0) {n = n - c;  x = y;}
          c = c >>> 1;
    } while (c != 0);
    return n - x;
}
//------------------------------------------------------------------------
// Reversing the rightmost 6 bits in a word.
 public static int rev3(int x) {
          return (x*0x00082082 & 0x01122408) % 255;
 }
//----------------------------------------------------------------------
}
```

2 Jude.java File

```
package genfiles;
public class jude {
 public static void main(String[] args) {
 }
 private int id;
 public jude(){id = 4;}
 public static boolean printString(){
  System.out.println("This needs to work");
  return true;
}

public static boolean echoString(){
 System.out.println("Come on baby");
 return true;
}

public static void pString(){
 System.out.println("This needs to work");
}

public static void eString(){
 System.out.println("Come on baby");
}
public static void p1(String s){
 System.out.println(s);
}

public static void p2(String str){
  System.out.println(str);
}

public static boolean d1(String s){
 System.out.println(s);
 return true;
}

public static boolean d2(String str){
 System.out.println(str);
 return true;
}

}
```

3 MultiClassTest.java File

```
package genfiles;
import java.awt.Rectangle;
public class MultiClassTest {
}
class Assignment2 {


  public int aTotal(int a, int b, int c){
        return a+b+c;
  }
//-----------------------------------------------------------------------
public String stringReverse(String source) {
 int i, len = source.length();
 StringBuffer dest = new StringBuffer(len);
 for (i = (len - 1); i >= 0; i--){
  dest.append(source.charAt(i));
 }
 return dest.toString();
}
//----------------------------------------------------------------------
//code example from http://www.leepoint.net/notes-java/data/arrays
   /31arrayselectionsort.html
public int[] sortInts(int[] x){
  for (int i=0; i<x.length-1; i++) {
   for (int j=i+1; j<x.length; j++) {
    if (x[i] > x[j]) {
      int temp = x[i];
      x[i] = x[j];
      x[j] = temp;
   }
 }
}
 return x;
}
//------------------------------------------------------------
//code example from http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html
public int[] BetterSort(int[] x) {
 for (int i=0; i<x.length-1; i++) {
 int minIndex = i;      // Index of smallest remaining value.
 for (int j=i+1; j<x.length; j++) {
  if (x[minIndex] > x[j]) {
        minIndex = j;  // Remember index of new minimum
  }
 }
 if (minIndex != i) {
 //... Exchange current element with smallest remaining.
  int temp = x[i];
  x[i] = x[minIndex];
  x[minIndex] = temp;
 }
}
 return x;
}
//------------------------------------------------
public double ConvertTempFtoC(double temp){
 return ((temp - 32)/9.0) * 5;
}
//----------------------------------------------
public double TempConverter(double t){
 double tmp = t - 32;
 tmp = tmp/9;
 tmp*=5;
 return tmp;
}
//----------------------------------------------
public class MyInner{
  public void innerM1(){}
//-------------------------------------
  public String innerM2(int i, String s){
        return "abcd";
  }
//----------------------------------------------
}
```

```
//--------------------------------------------------
public static class Innner2{
  public String newInner(int i, String y){
        return "Hellow";
  }
  public static void doNothing(int x){}
                }
//---------------------------------------------------
public String ForeCast(int temp){
  if (temp > 90 ){
        return "It is steamy!";
  }
  else if(temp > 80){
        return "It is hot!";
  }
  else if(temp > 70){
        return "It is great!";
  }
  else if(temp > 60){
    return "It is cold!";
  }
  else{
    return "It is fidge!";
  }
}
```

```
//———————————————————————————————————————
public String ArrayString(int[]x){
  StringBuilder sb = new StringBuilder();
  if(x.length > 0){
        for(int i = 0; i < x.length − 1; i++){
         sb.append(Integer.toString(x[i])+ ",");
        }
        sb.append(Integer.toString(x[x.length − 1]));
  }
  return sb.toString();
}
//———————————————————————————————————————
public int[] TestArrays(){
  int[] x = {1,2,3};
  return x;
}
//———————————————————————————————————————
public int[] Test2(){
  int[] x = {1,2,3};
  return x;
}
//———————————————————————————————————————
}
class Calculator3 {
 private int balance;
 public Calculator3(){
        balance = 0;
  }
//------------------------------------------------
public Calculator3(int arg){balance = arg;}
//-------------------------------------------------
public int getBal(){return balance;}
//---------------------------------------------
public int Add(int a){return a + getBal();}
//-----------------------------------------------
public int Sub(int a){return a - getBal();}
//------------------------------------------------
public int BalanceBurp(int a, int b, int c){
 return getBal()- a + a - b + c + b - c;
}
//-----------------------------------------------
public int Avg(int a, int b){
        return (a + b + getBal())/3;
}
//----------------------------------------------
public int returnParam(int a){return a;}
//---------------------------------------------
public int Zero(){return 0;}
//---------------------------------------------
public int square(){return 4 * 4;}
```

```
//----------------------------------------------------------------
public int Diff(int x){return x - balance;}
//-------------------------------------------------------------------
public int MinusSelf(){return balance - balance;}
//----------------------------------------------------------------
public int Identity(){return balance * 1;}
//----------------------------------------------------------------
public int TimesZero(){return balance * 0;}
//----------------------------------------------------------------
public boolean equal(Calculator3 c){
        return balance == c.balance;
}
//----------------------------------------------------------------
public boolean Same(Calculator3 p){
 boolean ret;
 if(balance == p.balance){
        ret = true;
 }
 else{
        ret = false;
 }
 return ret;
}
//----------------------------------------------------------------
public Calculator3 DoubleCalculator(){
        return new Calculator3(balance*2);
}
//------------------------------------------------
public Calculator3 NewCalculator(){
        Calculator3 c = new Calculator3();
        c.balance = balance *2;
        return c;
}
//------------------------------------------------
public String toString(){
        return Integer.toString(balance);
}
//------------------------------------------------
public Calculator3 MergeCal(Calculator3 c){
  return new Calculator3(balance + c.balance);
}
//------------------------------------------------
public Calculator3 AddCal(Calculator3 c){
 Calculator3 result = new Calculator3(0);
 result.balance = balance + c.balance;
 return result;
}
//------------------------------------------------
public Rectangle MyRectangle(int x, int y){
 return new Rectangle(x, y);
```

```
}
//---------------------------------------------
public Rectangle OutlineCalculator(){
 return new Rectangle(balance, 2*balance);
}
//---------------------------------------------
public Rectangle CalculatorRectangle(){
 int width = balance;
 int height = 2 * balance;
 Rectangle r = new Rectangle(width, height);
 return r;
}
//---------------------------------------------
public int[] TestArrays(){
 int[] x = {1,2,3};
 return x;
}
//--------------------------------------------
public int[] Test2(){
 int[] x = {1,2,3};
 return x;
}
//--------------------------------------------
public String reverseIt(String source) {
 int i, len = source.length();
 StringBuffer dest = new StringBuffer(len);
 for (i = (len - 1); i >= 0; i--){
  dest.append(source.charAt(i));
 }
 return dest.toString();
}
//------------------------------------------------
public int MyTotal(int a, int b, int c){
 return a+b+c;
}
}
```

4 MyTest.java File

```
package genfiles;
public class MyTest {
        private int id ;

        public MyTest(int x){
                id = x;
        }

        public static void getProduct(){
                System.out.println("hello world");
        }

        public static void anotherMethod(){
                System.out.println("Blah blah");
        }

        public int someMethod(){
                return id + 0;
        }

        public int getID(){
                return id;
        }
```

```
    public String toString(){
        StringBuilder sb = new StringBuilder();
        sb.append("id: "+ id + "\n");
        return sb.toString();
    }

    public boolean printString(String s){
        System.out.println(s);
        return true;
    }

    public boolean echoString(String str){
        System.out.println(str);
        return true;
    }
}
```

5 Shape.java File

```
package genfiles;
public class Shape {
 int numsides;
 int length;
 int width;
 boolean quad;

public Shape(int n, int l, int w){
 numsides = n;
 length = l;
 width = w;
 quad = isquad();
}
//---------------------------------------------------------------
private int area(){return length * width;}
//---------------------------------------------------------------
private int perimeter(){
 int tmp1 =  2*length;
 int tmp2 = 2* width;
 int res = tmp1 + tmp2;
 return res;
 }
//---------------------------------------------------------------
```

```
private boolean isquad(){return (numsides == 4);}
//---------------------------------------------------------------
private void enlarge(int howbig){
 length = length * howbig;
 width = width * howbig;
}
//---------------------------------------------------------------
private Shape copyshape(){
 Shape s =  new Shape(numsides, length,width);
 return s;
}
//---------------------------------------------------------------
private void maximize(int x){
 width = width * x;
 length = length * x;
}
//--------------------------------------------------------------
private int sizeAround(){return (2*(length+width));}
//--------------------------------------------------------------
public Shape  merge(Shape s){
 Shape tmp = new Shape(0,0,0);
 tmp.length = length + s.length;
 tmp.width = width + s.width;
 tmp.numsides = numsides + s.numsides -1;
 tmp.quad = tmp.isquad();
 return tmp;
}
//-----------------------------------------------------------
public Shape join(Shape s){
 Shape tmp = new Shape(0,0,0);
 tmp.length = length + s.length;
 tmp.width = width + s.width;
 tmp.numsides = numsides + s.numsides -1;
 tmp.quad = tmp.isquad();
 return tmp;
}
//------------------------------------------------------------
public void DoubleSize(){maximize(2);}
//------------------------------------------------------------
public void growTwice(){enlarge(2); }
//-----------------------------------------------------------
public String toString(){
 StringBuffer sb = new StringBuffer();
 sb.append("numsides: " + numsides + "\n");
 sb.append("length: " + length + "\n");
 sb.append("width: " + width + "\n");
 return sb.toString();
}
//------------------------------------------------------------
}
```

6 SmallTest.java File

```
package genfiles;
import java.util.Random;
public class SmallTest {
  public static int isqrt1(int x) {
  //this method finds the nearest square root of x
   int x1;
   int s, g0, g1;
   if (x <= 1) return 0;
   s = 1;
   x1 = x - 1;
   if (x1 > 65535) {s = s + 8; x1 = x1 >>> 16;}
   if (x1 > 255)   {s = s + 4; x1 = x1 >>> 8;}
   if (x1 > 15)    {s = s + 2; x1 = x1 >>> 4;}
   if (x1 > 3)     {s = s + 1;}
   g0 = 1 << s;                // g0 = 2**s.
   g1 = (g0 + (x >>> s)) >>> 1;  // g1 = (g0 + x/g0)/2.
   while (g1 < g0) {           // Do while approximations
          g0 = g1;                 // strictly decrease.
          g1 = (g0 + (x/g0)) >>> 1;
   }
//   System.out.print("g0: " +g0+ " ");
   if(g0 < 0)
          return 0;
```

```
    else return g0;
  }
//------------------------------------------------------------------------------

  public static int findSquareRoot(int myNumber){
  //find the nearest square root of myNumber
    final double EPSILON = .00001;
    int  guess = 1;
    double root = Math.sqrt(myNumber);
    while (EPSILON < Math.abs(Math.pow(root, 2) - myNumber))
         {guess++;}
    return (int)root;
   }
//-------------------------------
  public static void main(String[] args){
   Random r = new Random();
         for(int i = 0; i < 100; i++){
          int val = r.nextInt();
          if(findSquareRoot(val)==isqrt1(val))
                 System.out.println(i +".   true      (" +
                    findSquareRoot(val)+ ", "+isqrt1(val)) ;
          else
                 System.out.println(i +".   FALSE      (" +
                    findSquareRoot(val)+ ", "+isqrt1(val)) ;
         }
    }
}
```

7 Results.txt File

```
I am entering main
Percentage Similarity: 100
94 methods were found before filtering


79 Methods to be tested after filter
21 equivalence classes of Methods to be tested after filter
Done completing with effects
I am out of main

                         Cloneclass 0:
__________________________________________________

genfiles.Assignment2.ConvertTempFtoC Location: lines 53 - 55,
   Clone size: 3 LOC


genfiles.Assignment2.TempConverter Location: lines 57 - 62, Clone
   size: 6 LOC
__________________________________________________
                         Cloneclass 1:
__________________________________________________

genfiles.jude.p1 Location: lines 25 - 27, Clone size: 3 LOC  T


genfiles.jude.p2 Location: lines 29 - 31, Clone size: 3 LOC  T
__________________________________________________
```

```
                    Cloneclass 2:
------------------------------------------------------
genfiles.MyTest.printString Location: lines 31 - 34, Clone size: 4
   LOC

genfiles.MyTest.echoString Location: lines 36 - 39, Clone size: 4
  LOC
------------------------------------------------------
                    Cloneclass 3:
------------------------------------------------------
genfiles.Calculator3.OutlineCalculator Location: lines 199 - 201,
  Clone size: 3 LOC

genfiles.Calculator3.CalculatorRectangle Location: lines 203 -
  208, Clone size: 6 LOC
------------------------------------------------------
                    Cloneclass 3:
------------------------------------------------------
genfiles.Calculator3.getBal Location: lines 128 - 128, Clone size:
   1 LOC

genfiles.Calculator3.Identity Location: lines 152 - 152, Clone
  size: 1 LOC
------------------------------------------------------
                    Cloneclass 5:
```

```
genfiles.Calculator3.MergeCal Location: lines 185 - 187, Clone
   size: 3 LOC

genfiles.Calculator3.AddCal Location: lines 189 - 193, Clone size:
    5 LOC

                        Cloneclass 6:

genfiles.Shape.merge Location: lines 43 - 50, Clone size: 8 LOC

genfiles.Shape.join Location: lines 52 - 59, Clone size: 8 LOC

                        Cloneclass 7:

genfiles.Calculator3.DoubleCalculator Location: lines 171 - 173,
   Clone size: 3 LOC

genfiles.Calculator3.NewCalculator Location: lines 175 - 179,
   Clone size: 5 LOC

            Cloneclass 8:

genfiles.Calculator3.equal Location: lines 156 - 158, Clone size:
   3 LOC
```

```
genfiles.Calculator3.Same Location: lines 160 - 169, Clone size:
   10 LOC
```

```
Number of Clones: 12
Number of Clone Classes: 9
Maximum Clone Size : 10 LOC
Maximum Clone Class Size : 2 clones
Minimum Clone Size: 1 LOC
Minimum Clone Class Size: 2 clones
Average Clone Size: 4.0 LOC
Average Clone Class Size: 2.0 clones

Length of program execution: 54.576 seconds
```

LIST OF REFERENCES

[1] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, “Challenges of the dynamic detection of functionally similar code fragments,” *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 299–308, 2012.

[2] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA ’09, (New York, NY, USA), pp. 81–92, ACM, 2009.

[3] R. Elva and G. Leavens, “Semantic clone detection using method IOE-behavior,” in *Software Clones IWSC, 2012 6th International Workshop on*, pp. 80 –81, june 2012.

[4] D. Kawrykow and M. P. Robillard, “Improving API usage through automatic detection of redundant code,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE ’09, (Washington, DC, USA), pp. 111–122, IEEE Computer Society, 2009.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Comparison and evaluation of clone detection tools,” *IEEE Transactions on Software Engineering*, vol. 33, pp. 577–591, September 2007.

[6] H. S. Warren, *Hacker’s Delight.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] M. Flatt, S. Krishnamurthi, and M. Felleisen, “A programmer reduction semantics for classes and mixins,”

[8] G. Hedin and E. Magnusson, “JastAdd: an aspect-oriented compiler construction system,” *Sci. Comput. Program.*, vol. 47, pp. 37–58, Apr. 2003.

[9] T. LaToza, “A literature review of clone detection analysis.” Online, Retrieved May 5, 2008 2005.

[10] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, “The development of a software clone detector.” http://www.cs.cmu.edu/ aldrich/courses/654-sp05/tools/latoza-clone-detection-05.pdf, 1995 Retrieved May 5, 2008.

[11] R. Komondoor and S. Horwitz, “Using slicing to identify duplication in source code,” in *In Proceedings of the 8th International Symposium on Static Analysis*, pp. 40–56, Springer-Verlag, 2001.

[12] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, *Reverse engineering*, ch. Pattern matching for clone and concept detection, pp. 77–108. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[13] J. Krinke, “Identifying similar code with program dependence graphs,” *Proceedings of the Eighth Working Conference on Reverse Engineering*, pp. 301–309, 2001.

[14] B. Baker, “On finding duplication and near-duplication in large software systems,” *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pp. 86–95, Jul 1995.

[15] J. H. Johnson, “Navigating the textual redundancy web in legacy source,” in *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, p. 16, IBM Press, 1996.

[16] B. H. Nicolas Juillerat, “An algorithm for detecting and removing clones in Java code.” http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.3829, 2006.

[17] J. H. Johnson, “Identifying redundancy in source code using fingerprints,” in *CASCON ‘93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pp. 171–183, IBM Press, 1993.

[18] H. Ding and M. H. Samadzadeh, “Extraction of Java program fingerprints for software authorship identification,” *Journal of System Software*, vol. 72, no. 1, pp. 49–57, 2004.

[19] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: local algorithms for document fingerprinting,” in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 76–85, ACM, 2003.

[20] R. Koschke, “Survey of research on software clones,” in *Duplication, Redundancy, and Similarity in Software* (R. Koschke, E. Merlo, and A. Walenstein, eds.), no. 06301 in Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[21] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, “Clone detection using abstract syntax trees,” *Software Maintenance, 1998. Proceedings. International Conference on*, pp. 368–377, Nov 1998.

[22] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, “Clone detection in source code by frequent itemset techniques,” *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pp. 128–135, Sept. 2004.

[23] W. S. Evans, C. W. Fraser, and F. Ma, “Clone detection via structural abstraction,” in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 150–159, IEEE Computer Society, 2007.

[24] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “Deckard: Scalable and accurate tree-based detection of code clones,” in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, (Washington, DC, USA), pp. 96–105, IEEE Computer Society, 2007.

[25] M. Gabel, L. Jiang, and Z. Su, “Scalable detection of semantic clones,” in *ICSE '08: Proceedings of the 30th international conference on Software engineering*, (New York, NY, USA), pp. 321–330, ACM, 2008.

[26] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, “Measuring clone based reengineering opportunities,” *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pp. 292–303, 1999.

[27] F. V. Rysselberghe and S. Demeyer, “Evaluating clone detection techniques from a refactoring perspective,” *ase*, pp. 336–339, 2004.

[28] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, “Extracting code clones for refactoring using combinations of clone metrics,” in *Proceeding of the 5th international workshop on Software clones*, IWSC ’11, (New York, NY, USA), pp. 7–13, ACM, 2011.

[29] M. F. Zibran and C. K. Roy, “Towards flexible code clone detection, management, and refactoring in ide,” in *Proceeding of the 5th international workshop on Software clones*, IWSC ’11, (New York, NY, USA), pp. 75–76, ACM, 2011.

[30] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, “Automated scheduling for clone-based refactoring using a competent ga,” *Softw. Pract. Exper.*, vol. 41, pp. 521–550, April 2011.

[31] R. Tairas and J. Gray, “Sub-clone refactoring in open source software artifacts,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC ’10, (New York, NY, USA), pp. 2373–2374, ACM, 2010.

[32] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler, “A novel approach to optimize clone refactoring activity,” in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO ’06, (New York, NY, USA), pp. 1885–1892, ACM, 2006.

[33] H. A. Basit and S. Jarzabek, “Detecting higher-level similarity patterns in programs,” in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 156–165, ACM, 2005.

[34] T. Bakota, R. Ferenc, and T. Gyimothy, “Clone smells in software evolution,” *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 24–33, Oct. 2007.

[35] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Evaluating the harmfulness of cloning: A change based experiment,” in *MSR ’07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, (Washington, DC, USA), p. 18, IEEE Computer Society, 2007.

[36] S. Jarzabek and Y. Xue, “Are clones harmful for maintenance?,” in *Proceedings of the 4th International Workshop on Software Clones*, IWSC ’10, (New York, NY, USA), pp. 73–74, ACM, 2010. 11 references.

[37] C. Kapser and M. W. Godfrey, ““Cloning considered harmful” Considered harmful,” in *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE ’06, (Washington, DC, USA), pp. 19–28, IEEE Computer Society, 2006. 30 references.

[38] K. Kontogiannis, “Evaluation experiments on the detection of programming patterns using software metrics,” *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pp. 44–54, Oct 1997.

[39] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on*, pp. 253–262, Oct. 2006.

[40] H. A. Basit, U. Ali, and S. Jarzabek, "Viewing simple clones from structural clones' perspective," in *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, (New York, NY, USA), pp. 1–6, ACM, 2011. 6 references.

[41] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Softw. Pract. Exper.*, vol. 37, no. 2, pp. 151–175, 2007.

[42] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, (Piscataway, NJ, USA), pp. 364–374, IEEE Press, 2012.

[43] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 654–670, Jul 2002.

[44] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, (Washington, DC, USA), pp. 78–87, IEEE Computer Society, 2010.

[45] I. Keivanloo, C. Roy, and J. Rilling, "Sebyte: A semantic clone detection tool for intermediate languages," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pp. 247–249, 2012.

[46] B. Baker, "On finding duplication and near-duplication in large software systems," *Second Working Conference on Reverse Engineering(WCRE '95)*, pp. 86 – 95, 1995.

[47] A. Salcianu and M. C. Rinard, "Purity and side effect analysis for Java programs," in *Verification, Model Checking and Abstract Interpretation*, pp. 199–215, 2005.

[48] C. Razafimahefa, *A study of side-effect analyses for Java.* PhD thesis, School of Computer Science, Mc. Gill Univeristy, Montreal, 1999. 81 pages.

[49] N. E. Wallen and J. R. Fraenkel, *Educational Research : A Guide to the Process*, ch. 7. L. Erlbaum., 2000.

[50] J. Gradecki and N. Lesiecki, *Mastering AspectJ: aspect-oriented programming in Java.* Java open source library, Wiley, 2003. 434 pages.

[51] "java.lang.instrument Java platform SE 6." `http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html`, Feb. 2012.

[52] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, (Washington, DC, USA), pp. 107–, IEEE Computer Society, 2001.

[53] E. Juergens, F. Deissenboeck, and B. Hummel, "Clone detection beyond copy & paste," in *Proc. of the 3rd International Workshop on Software Clones*, 2009.

[54] D. Workman, “On the analysis of semantic clones,” Technical Report CS-TR-09-04, University of Central Florida, Department of EECS,University of Central Florida,4000 Central Florida Blvd,Orlando, Florida, 32816, USA, May 2009.