JSCTracker: A Tool and Algorithm for Semantic Method Clone Detection Using Method IOE-Behavior

Rochelle Elva and Gary T. Leavens

CS-TR-12-07
October 15, 2012

Keywords: Automated semantic clone detection tools, semantic method clones, method IOE-behavior, clone detection, JSCTracker.
2012 CR Categories: D.2.0 [*Software Engineering*] General — languages, tools, JSCTracker.

Copyright © 2012, Rochelle Elva and Gary T. Leavens

Computer Science, University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida 32816, USA

JSCTracker: A Tool and Algorithm for Semantic Method Clone Detection
Using Method IOE-Behavior

Rochelle Elva and Gary T. Leavens

University of Central Florida,
Orlando, Florida, USA

Abstract. This paper presents a tool and algorithm for the automated detection of functionally identical Java methods, which we call semantic clones. The detection algorithm offers improvements in asymptotic computational complexity over existing algorithms. This is achieved by combining the benefits of static and dynamic analysis. Our static analysis of a method's type (return type and parameter list) and effects (persistent changes to the heap) provides a double pre-filter, to reduce the size of the candidate clone set to be evaluated by expensive dynamic tests. Together, these two types of analyses provide the information on a method's input, output and effects, collectively referred to as its IOE-behavior, which we use to identify semantic equivalence. The tool was tested on 6 open source Java projects ranging in size from about 17 kLOC to 78 kLOC. Our filters reduced the number of required dynamic tests by an average of 91%.

Keywords: Automated semantic clone detection tools, semantic method clones, method IOE-behavior, clone detection, JSCTracker

1 Introduction

Code duplication in software projects exists in one of two formats: representational or functional. These give rise to syntactic and semantic clones respectively. There is extensive work done on syntactic clones (see Bellon *et al.*'s survey in [1]), but comparatively less on semantic clones, although their existence in real world software is evident in the clone literature. For example, Kawrykow and Robillard [2] find as many as 405 valid semantic clones in 589 kLOC of commercial software written in C. In our work we identify 502 semantic method clones in the 1677 methods of the Java open-source project *doctorj*. In addition, most of the tools designed for clone detection are aimed at recognizing only structural similarity. 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[3].

A possible explanation for the little work done on semantic clones might be that the exact detection of semantic equality is an undecidable problem. It is

arguable though, that the fundamental motivation for the detection and removal of syntactic clones is because of their potential for maintenance problems due to their semantic equivalence; not just their structure. Thus, semantic clones pose some of the same threats such as duplication of errors and update inconsistency. This, coupled with work demonstrating that semantic clones do exist in real world code, provides the motivation for our research.

In the existing work on semantic clone detection, we are aware of only two tools in the literature [4, 5] for automated detection, another paper that uses manual identification [3], and a tool that applies the concept of functional similarity to identify missed API re-use opportunities [2]. All of these 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 mutation of static and instance fields. The omission of effects allows simpler computations, however, it can imply lower precision and higher incidence of false positives.

1.1 Our Contribution

We present a tool for the automated detection of functionally identical Java methods: semantic method clones. It can identify methods such as those in Figure 1 which have identical behavior but not the same syntactic form.

```
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 HighestPowerof2(int x){
  int tmp = x;
  int answer = 1;
  while(tmp > 1){
    answer = 2 * answer;
    tmp = tmp/2;
  }
  return answer;
}
```

Fig. 1. Two semantic clones modified from *Hacker's Delight* [6]

Our approach combines the benefits of static and dynamic analysis. Our static analysis of a method's type (return type and parameter list) and effect(persistent changes to the heap), provide a double pre-filter, to reduce the size of the candidate clone set to be evaluated by expensive dynamic tests. Together, these two types of analyses provide the information on a method's input, output and effects behavior, collectively referred to as *IOE-behavior* [7], which we use to detect semantic equivalence. We show that this reduces the dynamic test set by

about 91% on average, thus improving the overall efficiency of the algorithm. For example given a program with 50 candidate method-clones, our filtering method would reduce the number of required tests by on average 91%: less than 5 tests, while other algorithms need 50 tests (see Section 5.1).

The rest of this paper describes our algorithm in detail. It also describes a case study which demonstrates the computational efficiency of combining static and dynamic analyses of method IOE-behavior in clone detection, versus using dynamic analysis only. We also investigate the relative merit of using two types of static filters to incorporate method behavior and effects into the detection process.[1]

2 The Approach

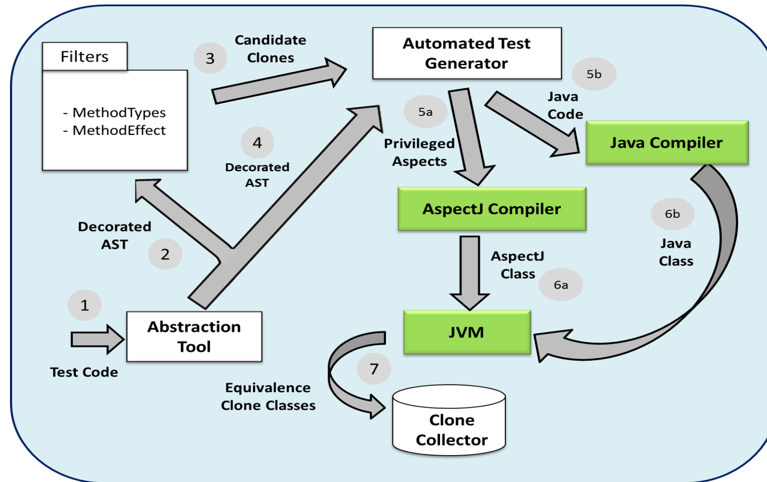

Fig. 2. JSCTracker architecture

Our approach analyzes Java code at the method level to detect functional equality. Our measure of functional equality is the input and total output behavior of a method; which is defined as the triad of its input, output and effects which we refer to as the method's *IOE-behavior*[7]. 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 also determined by two factors: the return value of the method (none for void methods), and its possible effects on the heap.[2] The latter refers to any mutations of instance or static fields which persist beyond the method call.

[1] A less detailed version of this algorithm with no experimental results is found in [7].
[2] Our effect analysis is a 'may' analysis referring to the possible effects of the method.

2.1 Overview of Algorithm

The semantic clone detection algorithm implemented by our tool can be summarized as follows:

Filtering using methodtype Group methods that have the same return type and the same corresponding type of parameters. Discard all groups containing only one method

Filtering using Effects Split groups into subgroups with the same *effects* - that is, they may write to the same fields. Discard all groups that have only one method

Testing For each group remaining after filtering using effects, for each method m in the group:

- **a** if m is an instance method, randomly generate a receiver object of m's receiver type. If m is static, skip this step.
- **b** If the *methodtype* of m has an empty parameter list, skip this step; otherwise generate a random instance of each parameter in the parameter list.
- **c** For instance methods, call m (using the receiver object from (a) and the parameters created in (b)). For static methods, call m using the parameters created in (b). Record the value returned; record the value and state of *effects* components listed in in the filtering with effects phase. Store result in the results array.[3]

Testing loop Repeat the testing phase a pre-determined numTests times

Collection Add m to a hashMap of results, hashed on the key of the results array. This automatically groups methods that have the same results for all of the test cases.[4]

Outputing report Discard hashMap rows for keys with only one associated method value. Output the rows of the hashMap that have a value field containing more than one method. These are the equivalence classes of semantic clones

The actual semantic clone detection occurs in 4 phases: abstraction, filtering, testing and collection, as shown in Figure 2 and discussed in the sections that follow.

2.2 Abstraction

The first phase, abstraction, depicted in steps 1 - 2 in Figure 2, uses a static analysis tool (built using the JastAdd compiler generator [8]), to produce a decorated abstract syntax tree. The abstraction encodes two types of information.

[3] All effects values and returned values (including exceptions) are converted to Strings to facilitate ease of comparisons.

[4] The use of the hashMap improves the efficiency of our algorithm, since the hash automatically groups methods with the same output values thus eliminating the need for additional comparisons.

The first is a *methodtype*. *Methodtypes* store a method's signature: parameter types, plus the type of the returned value. The second type of information is the method's *effects*: the set of fields that the method can write. This is represented as pairs of class and field names (both static and instance fields). To increase precision only write accesses are recorded in the effects. Table 1 demonstrates our properties of *methodtype* and *effects* for 3 methods of an account (Acct) class.

Table 1. MethodType and Effect Analysis for 3 Methods

Code	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}

2.3 Filtering

The filtering phase identifies preliminary sets of potential method clones, called the *candidate clone sets*, by applying two filters to the methods found. Both filters use static analysis of the source code.

The first filter: methodtype filter,uses the *methodtype* to group methods into equivalence classes. Only equivalence classes containing more than one method are considered for further analysis as clones. For example, given the six methods in Table 2, applying the first filter would result in 2 candidate clone sets: {`checkBal, calcInt`} and {`withdraw, deposit`}. The fifth 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 indicating deviating output behavior. The last two 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.

The second filter: methodtype + effects filter, uses static effects analysis on the output of the methodtype filter. This involves identifying possible writes on object parameters, static fields and instance fields of receiver objects (for instance methods) and writes to return values. Our analysis of the writes is an over approximation, since we also include possible writes such as those inside conditional statements, which may not occur at runtime. The methodtype + effects filter may split equivalence classes already created, or remove methods

Table 2. Sample Methods for Methodtype Analysis

Methods
double checkBal(){...}
double calcInt(){...}
void withdraw(double amt){...}
void deposit(double amt){...}
String getType(){...}
String printState(date d){...}

that share a type, but do not have the same effect. From the information shown in Table 3 for example, the second filter would result in only one candidate clone set: {`withdraw, deposit`}. The first candidate set will not be considered since `checkBal` and `calcInt` do not have the same effects.

Table 3. Candidate method-clones and their effects

Method	Effect
double checkBal()	{}
double calcInt()	{Acct.balance}
void withdraw(double amt)	{Acct.balance}
void deposit(double amt)	{Acct.balance}

The benefit of the two pre-test filters is to reduce the number of methods that require dynamic testing. This is an important contribution in terms of the complexity of the general algorithm, since dynamic testing may be arbitrarily costly.

2.4 Testing

The candidate clones are submitted as equivalence classes to the testing phase (step 3 in Figure 2). This phase acts as a dynamic filter comparing the actual behavior of methods when called in the same context. The context is built as heap states (parameter values) and object states (the value of receiver object fields and the value of static class fields — where relevant). The testing phase has two parts: generating a test file; and running it.

2.4.1 Generating the Test File. The generation of the test file is an automated process. It involves the creation of receivers and argument values for the methods to be tested. This process is discussed in detail below. Fundamentally, the generated test file consists of a series of method call statements where each method is called with different receiver objects and argument tuples.

2.4.1.1 Random Object Generation. Objects are generated using a call to a random constructor of the class, and passing randomly-generated parameter values where applicable. When the parameter is an object, a recursive process is used to generate the argument object.

For example, given the two candidate clone methods *withdraw*(*double*) and *deposit*(*double*) with a receiver object of type *Acct*, a uniform state is created for the two method calls by using the same randomly-generated double as the parameter and creating an object of the type *Acct* with randomly-generated values for each of the four fields. Some more interesting complications arise with abstract classes, and handling circular objects.

Abstract classes pose a peculiar problem since they cannot be instantiated. Our approach is to use instrumentation in Java to identify an instantiable subclass and generate an object of this class instead. This was made possible by using the Instrumentation API in Java 1.5. However, this process is costly so JSCTracker maintains a cached list of all already identified class substitutions to reduce the cost of subsequent searches. In cases where no class substitution could be found, the affected methods are not tested but the other members of the equivalence class may still be tested.

For circular objects we use a graph-coloring depth-first search algorithm to ensure that the creation of the object does not produce an infinite loop.

Another challenge was invoking private, candidate clone methods outside of their defining classes. To do this, we use privileged aspects in AspectJ. AspectJ was selected over editing the class code since it already offers the required functionality.

2.4.1.2 Automatically Generating the Test File. Once code for generating the required objects and states is created, a test file is automatically constructed containing the relevant method calls for each equivalence class. A matching privileged aspect is also generated to access private methods and fields.

It is our hypothesis that we can approach a good approximation to method semantic equivalence if we run the methods on a sufficiently large sample of their input domain—including both state and argument values.[5] Since only a sample of the heap state and object space can be tested in general, our algorithm runs multiple tests on objects in the same state and also runs each set of input parameter values with objects in multiple states. We call this our test-set. For example suppose the user-defined test-set size is 5. Then to test a method $A.b(c, d, e)$ we create 5 samples of objects of type A in different states. We also generate 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. Generally, we use a test-set size of 5 per method, since results did not vary between sizes 5–10.

2.4.2 Running the Test Files. The generated test file evaluates the dynamic behavior of the clone candidates, by using method calls to run each member of

[5] Our algorithm uses a timeout for method executions.

an equivalence class on the same input and then comparing the corresponding outputs. To ensure that each method is run on the same input we create a test-set for each type of receiver object. As the tests are run, each method is called with each pre-created receiver object and arguments in turn, and the output is recorded.

2.4.2.1 Determining Equivalence. A fundamental part of the dynamic testing for semantic clones, is determining equivalence. We define semantically equivalent methods as having identical IOE-behavior. Thus, our testing compares methods in a candidate set for equality using return value and effects. The comparison is trivial for primitive types. However, for values that are objects, we check that their final states are equivalent. The built-in Java method *Object.equals(Object)* does not always suffice; In some cases it returns a comparison of object addresses. For semantic clone detection though, we are interested in the equality of corresponding object fields. To guarantee this is the type of comparison done, we wrote a method which compares objects by recursively comparing the corresponding fields.[6]

2.5 Collection

This phase involves running the Test Driver as illustrated in (step 5b in Figure 2). 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. A method's results in this context is defined as the return values and the final states of all fields in method's effects.

3 Case Study

Our case study validates our tool using 6 relatively large samples of Java open-source software of different genres, shown in Figure3.

In this study we seek to answer three questions:

1. Can JSCTracker scale to automate the detection of semantic method clones in real-world Java code?
2. How does semantic clone detection using method type information compare to that of combining method type and effect information?
3. Do pre-testing filters using method IOE-behavior improve the efficiency of the clone detection process?

To answer these questions we analyze each of the above named samples of software using JSCTracker. For each sample the results are recorded for the time taken to complete test, the number of clones detected and reduction in the number of required tests brought about by the use of the two filters methodtype and effect.

[6] See section 4.2 for a discussion of how this may not always be right for all types of data

Source	Size in LOC	genre
Apache	19,499	Standard libraries
Doctorj	24,869	code analyzer
Freemind	16,450	mindmapping software
JabRef	74,586	reference management
Jetty	29,800	web-service software
JHotDraw	78,902	framework for creating drawing editor

Fig. 3. Source code used in case study

4 Results

			MethodType + Effects			MethodType			
Software	# Methods	# Equivalence classes	#clones found by MethodType + Effects Filter	# Clone classes	Execution time in sec for MethodType + Effects filter	#clones found by MethodType Filter	# Clone classes	Execution time in sec for MethodType filter	# Clones found in Deissenboek et al
Apache.lang3.3	313	49	50	13	0.33	46	11	0.29	55
doctorj-5.0	1677	63	502	9	3.44	893	30	0.31	na
freemind-2.6	4144	449	205	42	9.82	1435	198	1.38	11
JabRef-2.6	2269	133	54	11	19.14	720	95	3.6	55
jetty-6.1	1655	198	13	5	12.26	596	96	0.92	15
JHotDraw-7.0	1499	181	63	25	15.14	603	99	0.976	18
Avg	1926.17	178.83	147.83	17.50	10.02	715.50	88.17	1.25	30.80
Median	1666.00	157.00	58.50	12.00	11.04	661.50	95.50	0.95	18.00

Fig. 4. An overview of the results of the case study

Figure 4 is an overview of the results of our case study. On average we find 76 clones\kLOC. The median values are more representational of our findings though: 35 clones\kLOC. These method- clones belong to an average of 17, (or a median of 12) clone classes; indicating that all of these clones really represent 17(or 12) types of functional duplication in the code.

Figure 5 shows the number of semantic method clones identified in Deissenboeck *et al.* [5] and by JSCTracker using the methodtype filter and the methodtype + effects filter for each of the 6 software samples. It also gives the execution time for JSCtracker. In all cases except for JabRef and Apache, JSCTracker detects considerably more clones than Deissenboeck *et al.* (This is also evident from Figure 6.) It should be noted that doctorj is not tested in the work of Deissenboeck *et al.* [5].

A very large number of clones are detected using the methodtype filter only. This is explained by that filter's treatment of void methods. It uses an over-approximation for void methods. Since they do not have a return-type that can be compared, all void methods with the same parameter list are considered clones and are not tested dynamically; since there is no output value by which to measure equivalence. In the case of the methodtype + effects filter, the number of clones found is considerably smaller. With this combined filter, the void method-clone candidates are verified dynamically, since their output in terms of effects can be compared. However, even with this methodtype + effects filter, we detect a large number of clones for doctorj: 502, compared to less than 100 clones for all of the other software except for freemind, for which we detect 205.

Software	#clones found by MethodType + Effects Filter	Execution time in sec for MethodType + Effects filter	#clones found by MethodType filter	Execution time in sec for MethodType filter	# Clones found in Deissenboek et al
Apache.lang3.3	50	0.33	46	0.29	55
doctorj-5.0	502	3.44	893	0.31	0
freemind-2.6	205	9.82	1435	1.38	11
JabRef-2.6	54	19.14	720	3.6	55
jetty-6.1	13	12.26	596	0.92	15
JHotDraw-7.0	63	15.14	603	0.976	18
Avg	147.83	10.02	715.50	1.25	30.80
Median	58.50	11.04	661.50	0.95	18.00

Fig. 5. Clones detected

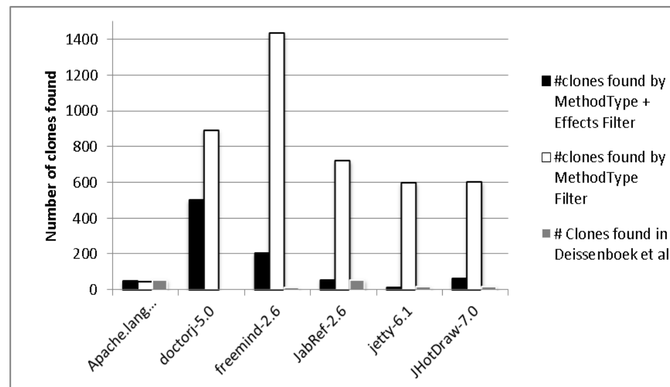

Fig. 6. Comparing the number of clones detected by the methodtype and methodtype + effects filters, with the work of Deissenboeck *et al.*

From Figure 5 it is evident that the methodtype filter is about 10 times faster than the methodtype + effects filter. However the later produces less false

positives since it can evaluate the equivalence of void methods. An interesting result is obtained for Apache, where their timing is only 0.04 seconds different and the methodtype + effects filter returns 4 more clones than the methodtype filter.

Figures 7 and 8 show the performance of the two filters in the reduction in the number of required tests. The best performance values are obtained for doctorj, while the lowest values are reported for Apache. Figure 8 shows that the filters succeed at improving the efficiency of the detection process, by reducing the number of required tests by as much as 84%-96%. These values confirm our hypothesis that the use of the filters would improve the overall efficiency of the algorithm.

Software	# Methods	# Equivalence classes after filters	% reduction in number of required tests
Apache.lang3.3	313	49	84
doctorj-5.0	1677	63	96
freemind-2.6	4144	449	89
JabRef-2.6	2269	133	94
jetty-6.1	1655	198	88
JHotDraw-7.0	1499	181	88
Avg	1926.17	178.83	90.72
Median	1666.00	157.00	90.58

Fig. 7. Equivalence classes after pre-filters

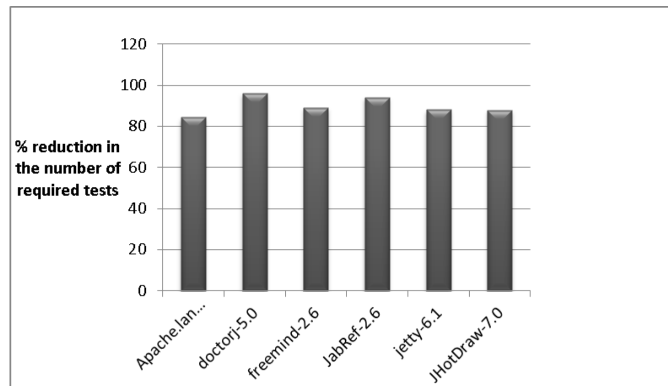

Fig. 8. Percentage reduction in the number of required testes using pre-filters

The methodtype filter is faster and in all but one case, identified more clones. A possible explanation for this is that the methodType filter accepts all void method clone candidates as clones without further testing. Another explanation is that the effects analysis filtered out some methods which should have been in the same equivalence class. Therefore our effects analysis rules might need to be refined as discussed in section 4.2.

4.1 Discussion

The number of clones JSCTracker detects is significantly greater than reported by Deissenboeck *et al.* [5] for the same software. There are many possible reasons for this. One reason could be the algorithmic differences and treatment of the code prior to testing. Deissenboeck *et al.* report that some files are eliminated during the detection process for technical reasons. Also, their study excludes constant methods and syntactic clones while we do not.

The large number of clones detected for doctorj was surprising. A Manual check of the detected clones however, revealed that they are quite semantically similar, but not equivalent. Interestingly, they are flagged as clones because it would require a large number of tests to simulate differences in their behavior. Many of these methods are 'switches' which access the same object fields and static variables; and more often than not, exhibit identical behavior. Thus, while 5–10 tests are enough to produce a fixed point in the set of clones for the other software samples, a much larger number of tests are required do the same for the doctorj code. The list does not converge even after a test size of 20: (which really means 20×20 tests, see Section 2.4.1.2). This also identifies a scaling issue with our tool. We are only able to run a test-size of up to 15 for doctorj, using the whole project as a unit.[7] We will address these issues in future work and investigate the benefits of using parallel programming.

4.2 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, we can use only a small fraction of it. 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 we could not test all cases our results can not be 100% accurate. In some cases, the randomly-generated objects are not appropriate, particularly when specially formatted data is required. For example with methods that require a file name or XML formatted string, our algorithm can generate a string, but the formatting test will fail and such methods will be bypassed for further testing. This issue can be addressed in future work by using preconditions in the random generation of objects.

[7] Tests are currently being run from within Eclipse. This itself causes problems with heap sizes for Eclipse.

In this first case study, we focus on improving the efficiency of the algorithm. However, recall and precision are also important in the evaluation of the effectiveness of clone detection tools. We will therefore address these in our further 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, prevent the algorithm from grouping these methods together. There is a similar issue with sub-typing. 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 super class 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. On the other hand, such methods are not substitutible for each other in Java's type system. Second, since or effects analysis is conservative, there is the possibility that methods are split into different groups that might not really differ in their write effects.

5 Related Work

5.1 Research on Semantic Clones

Recently there has been some interest in the detection of semantic clones although called by different names: "wide-miss clones," "high-level concept clones" [9], "functionally equivalent code" [5, 4], "behavioral clones" [10], "representationally similar code fragments" and "simions" [3]. These works seek to address the need to identify clones created by activities beyond mere copy and paste.

Juergens and Gode [3] investigate commercially used Java code JabRef. Manually inspecting 2,700 lines of code, they find that 32 out of 86 Utility methods are partially semantic clones—a little over 37%. In their manual approach, they do not consider entire methods but focused on functionally identical code fragments within different methods. Their comparison is not restricted to methods within the JabRef code. They also include similarity to Apache Commons methods. Our approach to semantic clone detection is different from theirs. Firstly, we present completely automated semantic clone detection, which makes the process objective. Secondly, our analysis is focused on the behavior of entire methods within the same body of code. We select this level of granularity because of the natural progression that it offers for code refactoring.

Jiang and Su [4] investigate functional similarity in code fragments within methods in the Linux system. The identified clone candidates are placed in equivalence classes and a representative member is selected. All other members of this class are then compared to the representative using dynamic testing. They found that about 42% of the semantic clones were also syntactic clones, thus the other 58% would be missed by syntactic clone detectors. They record low overall precision values due to the high number of false positives produced.

A possible drawback of this algorithm is that the quality of the outcome is greatly determined by the element of the cluster that is first selected to represent the group. Also, code effects are not included in the analysis of input-output behavior. We cannot compare their results to our work, since the two studies are so different. We study Java code, while they focus on the Linux system written in C. Our level of granularity was the method while theirs was a method fragment. Also, since our focus is on methods, which are naturally occurring executable syntactic units, our computations are simple and provide a less invasive and possibly less disruptive starting point for code maintenance through refactoring.

Another application of semantic clone detection is demonstrated by Kawrykow and Robillard [2]. They use functional similarity to identify instances of less than optimal usage of APIs, where developers re-implement library functions instead of making calls to an available API function. Using their detection tool, iMaus, they study 10 widely-used Java projects from the SourceForge repository. The projects ranged in size from 20 to 539 kLOC, finding 4 to 341 method imitations. 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.

Dissenboeck *et al.* [5] present an algorithm for the dynamic detection of functionally similar code fragments. 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 [11]. Our approach to semantic clone detection is different from theirs because we consider only semantic method clones and include method effects in the detection process. We chose to restrict the detection process to semantic method clones for multiple reasons. Firstly, methods are semantic units of Java code and are thus a natural unit for reuse. Thus, they have greater practical application for code maintenance through refactoring with the minimal disruption of the code. Chunks or method fragments are a less intuitive choice for refactoring, also the computation of their input and output values is complex. In addition, because these exist as sub-components of a bigger unit, extracting them and attempting to mimic the behavior is potentially faulty, since the behavior may be determined by the context. Our algorithms also differ in our input value generation. While Deissenboeck *et al.* [5] use all of the constructors for the methods found, to generate the test cases, we use a user defined number of randomly selected constructors with the same number of randomly-generated actual input values. Consequently, we produce a smaller yet still diverse subset of the entire test set. Our algorithm also combines static analysis to pre-filter the candidate set before dynamic testing thus reducing the actual number of methods to be tested.

6 Conclusion

The motivation for the research in this paper was to detect semantic clones in real-world software and to evaluate the impact of using pre-testing filters on the

efficiency of the clone detection process. We met both of these goals. JSCtracker detected as many as 502 clones in real-world software. From our results, it is also evident that pre-filters can contribute greatly to the improvement of the efficiency of the detection algorithm by reducing the required test size by about 91% on average.

In the future we will investigate the precision and recall of our methods and the possibility of developing a benchmark for semantic clones.

References

1. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering **33**(9) (September 2007) 577–591
2. Kawrykow, D., Robillard, M.P.: 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, IEEE Computer Society (2009) 111–122
3. Juergens, E., Deissenboeck, F., Hummel, B.: Code similarities beyond copy & paste. In: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering. CSMR '10, Washington, DC, USA, IEEE Computer Society (2010) 78–87
4. Jiang, L., Su, Z.: 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, ACM (2009) 81–92
5. Deissenboeck, F., Heinemann, L., Hummel, B., Wagner, S.: Challenges of the dynamic detection of functionally similar code fragments. Software Maintenance and Reengineering, European Conference on **0** (2012) 299–308
6. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
7. Elva, R., Leavens, G.: Semantic clone detection using method IOE-behavior. In: Software Clones (IWSC), 2012 6th International Workshop on. (june 2012) 80 –81
8. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Sci. Comput. Program. **47**(1) (April 2003) 37–58
9. Marcus, A., Maletic, J.I.: 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, IEEE Computer Society (2001) 107–
10. Juergens, E., Deissenboeck, F., Hummel, B.: Clone detection beyond copy & paste. In: Proc. of the 3rd International Workshop on Software Clones. (2009)
11. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 321–330