JSCTracker: A Semantic Clone Detection Tool for Java Code

Rochelle Elva and Gary T. Leavens

CS-TR-12-04
March 2012

Keywords: semantic clone detection, input-output behavior, effects, IOE behavior, Java language, JSCTracker.

2000 CR Categories: D.2.0 [*Software Engineering*] General — languages, tools, JSCTracker;

Department of EECS
University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida, 32816, USA

JSCTracker: A Semantic Clone Detection Tool for Java Code

Rochelle Elva
University of Central Florida
4000 Central Florida Blvd.
Orlando Fl., 32816
Email: relva@eecs.ucf.edu

Dr. Gary T. Leavens
University of Central Florida
4000 Central Florida Blvd.
Orlando Fl., 32816
Email: leavens@eecs.ucf.edu

***Abstract*—This paper presents a tool and algorithm for the detection of semantic clones in Java methods. For our purpose, semantic clones are defined as functionally identical code fragments. Thus, our detection process operates on the premise that if two code fragments are semantic clones, then their input-output behavior would be identical. We adopt a wholistic approach to the definition of input-output behavior by including not only the return values of methods; but also their effects as reflected in the pre- and post-states of the heap. We refer to this as a method's *IOE*(input, output and effects)-*behavior*. Our tool and algorithm are tested in a small case study using the open source database management software *DSpace* and the reference management software *JabRef*.**

***Keywords*-Software Clone Detection, Semantic Clones, Semantic Clone Detection Tool, Method IOE-behavior, Program Understanding**

I. Introduction

Duplication in code exists in one of two forms: representational or functional. The former case yields syntactic clones, while the latter is responsible for the formation of semantic clones. Although current literature indicates that as many as 405 valid semantic clones can be found in as few as 1589 kLOC of commercial software written in C [1], the majority of clone detection algorithms and tools are limited to the detection of syntactic clones. For example, when tested on 109 variations in implementations of the same functionality, 4 state of the art tools were only able to identify 1% of the semantic clones [2]. The reason for this might be that the detection of semantic similarity is an undecidable problem. However, since the presence of semantic clones can cause some of the same maintenance issues as their syntactic counterparts [3] they should not be ignored; thus even a good approximation of their detection is valuable.

Another deficiency in existing clone detection strategies is that they fail to guarantee an adequate level of precision and thus are not accepted or implemented by industry [4]. According to the literature, such tools report as much as 27% - 65% [5] of false positives[1]. Some claim that the primary reason for this low precision is the lack of clear and standardized criteria for distinguishing between clones and non-clones. Even when benchmarks are used, the criteria are so subjective that the results are unreliable and inconsistent. In one study for instance, when asked to examine 317 candidate clones, only 5 were categorized in the same way by three judges [4].

In the limited work that has been done on semantic clone detection, we could only find one tool in the literature [6] for automated detection, and another paper which used manual identification [2]. Both of these used input-output as the technique for identifying semantic clones. However, this technique is incomplete. The behavior of a method also includes its effects, which have been ignored in both of these papers. While this omission lends to simpler computations, it contributes to the problems of lower precision and higher incidence of false positives.

In this paper we address these problems by outlining the following research goals:

1) To define semantic method clones in objective, concise and comprehensive terms
2) To automate the detection of semantic method clones using a combination of their *IOE-behavior*
3) To develop a semantic method clone detection algorithm and tool, with precision values that are an improvement over the 27%-65% reported for existing tools and methods.

II. The Approach

Our approach works at the method level in Java code. We define functional similarity in terms of input and total output behavior. Input, is determined by two factors, namely: the value of parameters passed to the method, and state of the heap when the method is invoked. Output on the other hand, is defined in terms of both the method's return values and its *effects*. The former is the value returned by the method as represented by the return type in the method's signature. This value is ignored for *void* methods. The latter represents the possible, persistent changes to the heap as a result of the execution of the method.[2] These changes include mutations

[1]False positives are code fragments output as clones when actually they are not

[2]Throughout this paper when we speak of a method's effects, this will refer to the set of possible effects (which is a "may" analysis in the jargon of static analysis).

Table I
SAMPLE METHODS FOR METHODTYPE AND EFFECT ANALYSIS

Method	Code	MethodType	MethodEffect
Balance	`double checkBal(){return balance;}`	double,void	{acct.balance}
calcInt	`double calcInt(){return balance * rate;}`	double, void	{acct.balance, acct.rate}
PrintState	`String PrintState(date d){return "The balance to date is: " + balance;}`	String, date	{acct.balance}

Table II
SAMPLE METHODS FOR METHODTYPE ANALYSIS

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

Table III
CANDIDATE METHOD CLONES AND THEIR EFFECTS

Method	Effect
double checkBal(){...;} double calcInt(){...;}	{acct.balance} {acct.balance, acct.rate}
withdraw(double amt){...;} deposit(double amt){...;}	{acct.balance} {acct.balance}

of static fields of classes and data members of objects. All of this behavior we refer to collectively as a method's *IOE-behavior.* (input, output and effects behavior).

The actual semantic clone detection involves 4 sub-processes: abstraction, filtering, testing and collection, as shown in Figure 1 and discussed in the sections that follow.

A. Abstraction

The first phase, abstraction, depicted in steps 1 - 2 in Figure 1, uses a static analysis tool along with the JastAdd compiler generator, to produce a decorated abstract syntax tree (AST). This is achieved by defining parsing rules and using a combination of aspect-oriented programming and attribute grammar concepts to define program properties. These properties are later used to create an abstract representation of the methods of the code parsed. The abstraction encodes two types of information. The first is a *methodtype*. *Methodtypes* are represented as Java objects with data members used to store a method's signature and parameter types, plus the type of the returned value. The second are its *effects* represented as pairs of class and field names (both static and instance fields). This produces an over-approximation of the method effects, since any access of the variable types just described is treated as a possible mutation. Table I demonstrates our properties of *methodtype* and *effects* for 3 methods of an account (acct) class.

B. Filtering

As shown in step 2 of Figure 1, the filtering phase uses the code abstractions to identify preliminary sets of potential method clones, called the *candidate clone sets*. Two filters are applied to the identified methods.

The first filter uses syntactic information encoded in the *methodtype*. After this filter has been applied, only methods with correspondingly equivalent return types and parameter types will be considered *candidate clones*. For example, given the six methods in Table II, 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; 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 uses semantic information in the form of method effects. Using the *effects* properties of the decorated AST nodes, the candidate method clones returned by the first filter are further reduced by comparing their *effects*. From the information shown in Table III for example, the second filter would result in only one candidate clone set - {`withdraw, deposit`}. The first candidate set will no longer be considered since `checkBal` and `calcInt` do not have the same *effects*.

The net result of the two filters, is that the only methods left in any equivalence class, would have the same *methodtype* and *effects*. The reduction in the number and sizes of equivalence classes is important, as it decreases the number of computations at the next level of the detection process, which is computationally the most costly.

C. Testing

The problem of determining if two methods are semantically equivalent is undecidable. However, it is our hypothesis that we can approach a good approximation if we run the methods on a sufficiently large sample of their input domain - (this includes state and values).[3]

The candidate clones are submitted as equivalence classes to the testing phase (step 3 in Figure 1). A test file is automatically generated for each set (step 5a in Figure 1). This file facilitates the evaluation of the actual 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. To ensure that each method is run on the same input we create a test-set of each type of receiver object. As the tests are run,

[3]Our algorithm uses a timeout for method executions

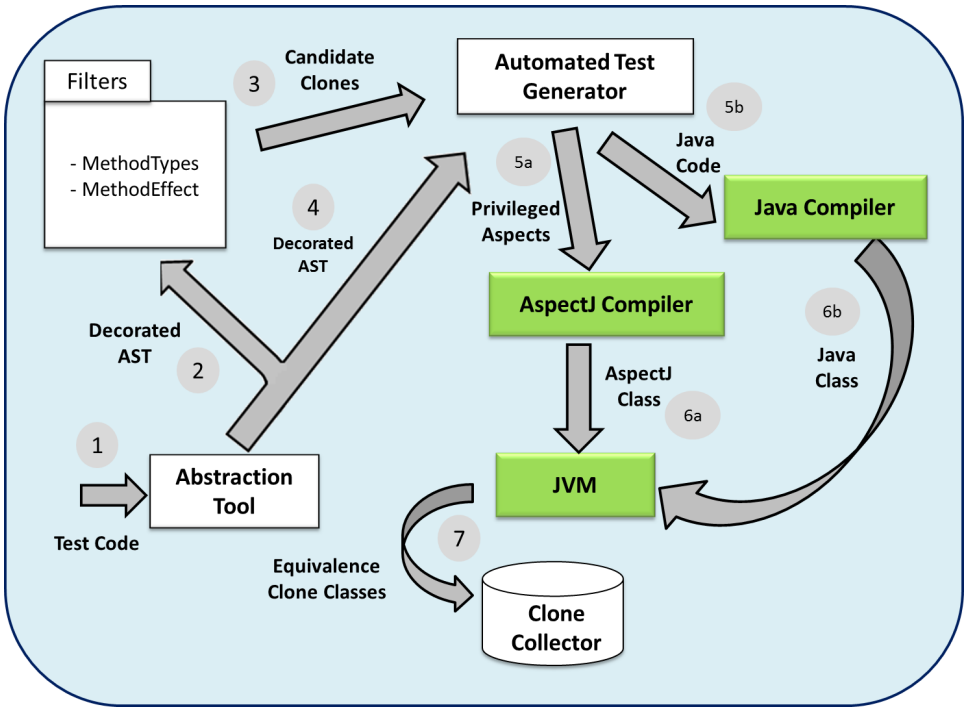

Figure 1. JSCTracker Architecture

each method is called with each pre-created receiver object in turn, and the output is recorded. Since there is potentially an infinite number of possible combinations of legitimate object states and parameter values, and only a finite set of those can practically be evaluated, randomly generated values are used for each of the datamembers of the object, to explore the space of the inputs. The reliability of the results is enhanced by creating a user-defined number of test cases. Generally we used five, since our empirical tests showed that the results did not vary when 5–10 tests were selected. Thus, in the interest of time efficiency, we selected the lower bound.

D. Collection

The testing phase also automatically generates a *Test Driver* file to run all of the test files (step 5b in Figure 1). 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 components of the method's *effects*. In addition, the members of any equivalence class output, would be of the same *methodtype* and thus they would have the same observable *IOE-behavior*. The driver also outputs a statistical summary of the clone data including: the number, location and size of clones and clone classes. Minimum, maximum, and average values are also provided. All output data is collected and stored in the clone collector (as shown in step 7 in Figure 1).

E. Semantic Clone Detection Algorithm

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

1. Group methods that have the same return type and the same corresponding type of parameters. Discard all groups containing only one method
2. For each group of methods left after step 1, group methods that have the same *effects* - that is, they may write to the same fields. Discard all groups that have only one method
3. For each group remaining after step 2, 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 step 2. Store result in results array.[4]

d Repeat step 3 a pre-determined numTests times

e 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.[5]

5 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

III. Related Work

A. Clone Detection Techniques

Over the last two decades, software clone research has been growing in importance. Although at one time dubbed the cause of bad smells in code [7], this perception is now evolving. There is a growing trend in the software clone research community, to accept that the presence of clones in code is not necessarily as bad as previously thought [8]. However, detection is still viewed as important to program understanding and maintenance. This has sparked a change in the direction of research from absolute elimination to more accurate detection, monitoring and interactive refactoring.

The research on clone detection has lead to five general categories of algorithms based on: strings [9], [10]; tokens/lexicons [9], [11], [12]; abstract syntax trees (AST) [13], [14], [15], [16]; program dependence graphs (PDG) [17], [18] and metrics [19], [20], [21], [10], [22], [23], [24]. A variety of combinations of these are also used in hybrid techniques. However, all of these focus on syntactic clones. Our contribution to this area of research is to join the few who have begun to venture into the detection of semantic clones and explore its relevance to software maintenance.

B. Research on Semantic Clones

The detection of semantic clones is an interesting area of research. Not only does it facilitate a more complete picture of the code duplication in software, but it also identifies the clones that are the most obvious and useful candidates for code improvement through refactoring. Recently there has been some interest in the detection of semantic clones although called by different names: wide-miss clones, high-level concept clones [25], functionally equivalent code [6], behavioral clones [26], representationally similar code fragments and *simions* [2], they all seek to address the need to identify clones created by activities beyond mere copy and paste.

Juergens and Gode [4] investigated commercially used Java code JabRef and out of 2,700 LOC they manually found that 32 out of 86 Utility methods were partially semantic clones - a little over 37%. In their manual approach, they did not consider entire methods but focused on functionally identical code fragments within different methods. Their comparison was also not restricted to methods within the JabRef code. They also included similarity to Apache Commons methods. Our approach to semantic clone detection is different to theirs. Firstly, we improve on the objectivity of the detection process by presenting a completely automated semantic clone detection. Secondly, our analysis is focused on the behavior of entire methods within the same body of code. We selected this level of granularity because of the natural progression that it offers for code refactoring.

Jiang and Su [6] investigated functional similarity in code within methods. Their evaluation was based on input:output behavior while ignoring effects. In studying the Linux system, they found that a little more than 42% of the semantically similar code was also syntactically similar; while more than 36% of syntactic clones are not semantic clones. The overall precision is low due to the high number of false positives produced because effects are ignored. Also clone candidates are not whole syntactic units like methods so inputs and outputs have to be computed. In computing equivalence for each cluster they selected a representative, then ran sample code on that member and recorded the output. Each member of that equivalence class was then run on the same input and the output and then compared to the result obtained from the representative one. One issue with this algorithm is that the quality of the outcome is greatly determined by the element of the cluster that was first selected to represent the group. While our algorithm in its current state does not scale as well as this one, our results have greater precision since clones are not selected on the basis of comparison to an arbitrarily selected candidate. Instead the behavior of each method is considered and subgroups can be identified by use of hashing on the results. Also, since our focus is on methods which are naturally occurring syntactic units, our computations are simpler 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 [1]. They use semantic clone detection to identify instances of less than optimal usage of APIs, where developers re-implemented library functions instead of making a call to an available function. Using their detection tool iMaus, they studied 10 widely used Java projects from the SourceForge repository. The projects ranged in size from 20 to 539 kLOC finding 4 to 341

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

[5]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.

method imitations with the median precision being 14.5%.[6]

We present a very simple method for detecting semantic clones. Our focus is on the *IOE-behavior* of methods. We simulated this behavior using several test cases, from the input domain of the code and then analyzing the resulting output behavior. One other contribution of our algorithm is that we do not employ any computationally complex code abstractions as found in [1], [6].

IV. Case Study

Our tool was pilot tested on code written by one of the authors. This included 22 classes ranging in size from 900 lines of code (LOC) and 39 methods; to 70 LOC with 12 methods. These small projects were used to test the algorithms robustness for identifying different categories of semantic clones - from the syntactically identical to the very structurally divergent yet functionally equivalent code. For the latter type of clone, we modified code found in *Hacker's Delight* [27] and created methods such as `flp2` and `HighestPowerof2` shown in Figure 2. Both of these methods return the highest power of two that is less than or equal to some input integer. Yet, this is not apparent just by looking at them. Our tool was able to identify all 13 of the semantic clones created from code adapted from *Hacker's Delight*, with 100% precision and recall.

Since we use methods, as our level of granularity, we also explored both static and instance methods with a range of primitive and object return types and parameters, and arrays of the same. We were able to identify the semantic clones with 100% precision and recall (thus there were no false positives).

In our actual case study we tested 1 package of the open source database management software *DSpace* and 3 from the Java Bibtex Reference software *JabRef*. The results are shown in Table IV.

A. Results

We found no semantic clones in the DSpace software tested and four clones in one of the JabRef packages - gnu.tools.ritopt. After the filtering phase for this package, our tool identified 16 equivalence classes of candidate clones. This was reduced by the testing phase to four clones in two equivalence classes as shown in Table IV. On examination of the code, we found that these were actually semantic clones. None of the other JabRef packages tested yielded any clones. All of the pakages tested had execution times from 0.016–0.93 seconds.

The number of clones that JSCTracker found was less than that recorded by Juergens and Gode [4] in the same software - JabRef. However, as already discussed in section III-B of the related work, the two approaches to semantic clone detection were different. While we focused only on the JabRef code, they included similarities between JabRef and Apache. They also included partial methods (method fragments) in their clone count, while we looked for semantic similarity between complete methods.

[6]The mean was 31% but the median gives a more true picture of the overall precision of the system.

```
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;
}
```

Figure 2. Semantic Clones from Hacker's Delight

B. Limitations

This is a preliminary study on the effectiveness of JSCTracker, as such one of the primary limitations is the size of the case study. In terms of our algorithm, two of its limitations are scalability and the use of only an approximation of the input space of the tested methods. Since the input space is possibly infinite in size, we need to obtain some finite sample set. To make this set representative of the whole, samples were generated at random using an unbiased process to ensure that all cases were equally likely. However, because we could not test all cases our results would not be 100% accurate.

In this first case study, our focus was on precision. Thus we were primarily concerned with reducing the number of false positives. We do acknowledge, however, that recall is also important in the evaluation of the effectiveness of clone detection tools. We will therefore address this issue more, in our further work. The merit of good recall values was not totally ignored though, as we were able to produce recall values for the author's code (which was 100%), for which it was easy to create the clone corpus.

Our algorithm's filters which are intended to improve efficiency, present some possible threats to *recall*. First, 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

Table IV
RESULTS FROM RUNNING JSCTRACKER ON COMMERCIAL SOFTWARE

Software	Package	#Classes	#Candidate Clone Classes	Clone Classes	#Clones	Time(sec)
Jabref	gnu.dtools.ritopt	35	16	2	4	0.850
Jabref	net.sf.jabref.undo	11	4	0	0	0.016
Jabref	net.sf.jabref.about	6	7	0	0	0.023
DSpace	dspace.administer	12	16	0	0	0.093

grouping these methods together. Second, the effect analysis, since it's overly conservative, would split methods into different groups that might not really differ in their write effects (but only, in what they read). However, testing won't affect recall, since it only splits apart method groups, it doesn't prevent them from forming. And when tests show that two methods aren't semantic clones, then the tests are right.

V. CONCLUSION

We began our study attempting to address three issues, namely: providing a concise and objective definition for semantic clones, automating the detection process and producing an algorithm with precision values greater than the average 27%-65%. In this paper we have met these goals for code of limited size. We have presented a very objective definition of semantic clones using the *IOE-behavior* of code. This leaves little room for personal interpretation. This preliminary study of JSCTracker is encouraging in its ability to identify semantic clones. Our test cases though small, returned 0% of false positives. While we are able to say that the recall was 100% for the author developed code, we need to continue testing to find its recall for other larger projects. It's not impossible for our algorithm to return false positives. However, it is our hypothesis, that the likelihood of false positives can be decreased by running more test cases in the *Testing* phase. In the future we will continue to develop and test our tool to improve scalability and to evaluate its effectiveness in both precision and recall.

REFERENCES

[1] 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*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122, this paper has 21 references. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.62

[2] B. H. E. Juergens, F. Deissenboeck, "Code similarities beyond copy & paste," in *CSMR '10: Proc. of the 14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain, March 2010, p. 10.

[3] V. R. Richard Fanta, "Removing clones from the code," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 4, pp. 223–243, July/August 1999, 17 pages.

[4] E. Juergens and N. Göde, "Achieving accurate clone detection results," in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10. New York, NY, USA: ACM, 2010, pp. 1–8, there are 17 references in this paper. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808902

[5] C. Kapser and M. W. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings of the Principles of Software Evolution, 7th International Workshop*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 85–94, this paper has 23 references. [Online]. Available: http://portal.acm.org/citation.cfm?id=1018436.1021754

[6] 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*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 81–92, this paper has 39 references. [Online]. Available: http://doi.acm.org/10.1145/1572272.1572283

[7] 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.

[8] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 19–28.

[9] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," *icpc*, vol. 0, pp. 153–162, 2008.

[10] 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*. IBM Press, 1993, pp. 171–183.

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

[12] 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: ACM, 2005, pp. 156–165.

[13] 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: IEEE Computer Society, 2007, pp. 150–159.

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

[15] 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.

[16] 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.

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

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

[19] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," *Software Maintenance 1996, Proceedings., International Conference on*, pp. 244–253, Nov 1996.

[20] F. Nielson and H. R. Nielson, *Correct System Design*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, January 1999, vol. 1710/1999, ch. Type and Effect Systems, pp. 114–136.

[21] 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.

[22] 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.

[23] 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: ACM, 2003, pp. 76–85.

[24] 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.

[25] 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*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 107– , this paper has 46 references. [Online]. Available: http://portal.acm.org/citation.cfm?id=872023.872542

[26] E. Juergens, F. Deissenboeck, and B. Hummel, "Clone detection beyond copy & paste," in *Proc. of the 3rd International Workshop on Software Clones*, 2009, there are 4 references in this paper.

[27] H. Warren, *Hacker's delight*. Addison-Wesley, 2003. [Online]. Available: http://books.google.com/books?id=iBNKMspIlqEC