Clone detection algorithm based on the Abstract Syntax Tree approach

Flavius-Mihai Lazar, Ovidiu Banias
Politehnica University of Timisoara, Timisoara, Romania
ovidiu.banias@aut.upt.ro

Abstract— **In this paper we present useful methodologies in analyzing program code written in C programming language in order to detect source code clones between two or multiple files. For detecting student plagiarism in C code, we implemented a three phase clone detection algorithm based on the AST (Abstract Syntax Tree) approach. Starting from the state of the art in this domain, we make a short description and comparison between the proposed methods and finally we present a case study upon student assignments for the Programming Techniques classes. In the end, we conclude pointing the future directions of development and improvement in order to obtain a better clone detection tool.**

I. INTRODUCTION

Code duplicates represent a common problem for all the software systems and especially for the large ones, in the same time being a sign of bad design [1]. The analysis of the software systems has proven that a considerable amount of the source code is duplicated. According to [2], 5-10% of the source code consists of duplicated code and according to [3] this percentage rises to 13-18%. The main reason for which code clones appear is the way the programmers understand to reuse code. They tend to directly copy fragments of code that implement something similar to their needs and by performing small changes they adapt it to the new context and that piece of code is reused. Another reason for which programmers duplicate code is presented in [4]: it is easier and faster to copy a code fragment (which may be already tested) than writing the code from scratch. As presented in [5], the occurrence of software clones has many different causes: reusing code by copying existing idioms, coding styles, instantiations of definitional computations, failure to identify or use abstractions, performance enhancement, accidents.

Most of the state-of-the-art software systems enjoy a structured design and formal ways of code reuse, but legacy code has unfortunately less structure. A considerable amount of code was produced reusing existing code through copy and paste. When implementing new functionality, programmers (also due to time pressure) search for some code idiom that implements a functionality almost identical to the one desired, then copy the idiom and adapt it to the new context. The performed changes are small and limited to renaming variables, inserting or deleting small code fragments or changing values of constants. This habit, may be a way of producing variant modules. For example, when building device drivers for different operating systems, an important part of the source code is identical between two versions, and the only part that contains hardware specific code requires changes. In such situations, the author will probably copy an existing, well-known, trusted driver and will modify it according to his particular specifications. In practice, this may be a good example of reuse, but in the case a bug is found in the trusted driver, fixing requires changes in all new device drivers having the trusted one as template.

When a set of simple computations (like data structure access) are used repeatedly, they became definitional and even when there is no code copying, programmers might use a mental macro to write essentially the same code each time. If the mental operation is used frequently, it may turn into a coding style for the programmer. Systems that require time constraints are often manually optimized by replicating frequent computations. In this case, code duplication is done only for performance enhancement reasons. Occasionally, code fragments are accidentally identical, but in fact they are not clones. As the code increases in size, the number of such accidents decreases dramatically [2].

Code duplication may have its justifications but in general it is considered bad practice and refactoring is recommended over the duplicated code. In [6] Martin Fowler introduced the code refactoring rule of thumb, named “Rule of three”, that is used to decide when a duplicated code should be extracted in a function. The rule allows to copy a piece of code only once, but when the code is copied the second time, (having now three identical pieces of code) it should be replaced by a function. Refactoring may require large amounts of knowledge about the architecture and design and thus creating an abstraction represents also a risk.

Code duplication is considered bad practice because it makes the code harder to maintain (and it is estimated that 70% of the effort put in developing a software system goes into the maintenance phase [4]). Sometimes, detection and analysis of code clones can be simple and straightforward, but in the majority of cases and especially for large software systems it is not an easy task. Making use of a powerful and reliable tool for such task of clone detection is very useful and quite mandatory when dealing with legacy systems [7].

Detecting plagiarism is another important application of the detection and analysis of the code clones algorithms. In this paper we propose a clone detection algorithm in order to detect plagiarism in multiple file sets.

II. CLONE DETECTION

The first step to be done in the war against code clones is to detect the existing clones in the program's source

978-1-4799-4694-5/14/$31.00 ©2014 IEEE

code. **After** analyzing the results from the first step, developers can decide whether or not is **necessary** to refactor the source code and to introduce abstractions that will replace the clones. This approach is not straightforward and represents a risk as the developer must have solid knowledge of the program's architecture and design; using a clone detection tool will ease a lot the developer's task of finding and analyzing code clones and will increase his productivity.

Clone detection algorithms can be categorized depending on how they handle the program's source code. Some of them perform a line of code analysis, which is simple and faster, but detect only exact clones and fail to detect near-miss clones. Other algorithms are based on programming language level tokens or even the abstract syntax tree (AST) and are able to detect also near-miss clones, with the drawback of being slower. Another type of clone detection algorithms are based on slicing [8], being the slowest but delivering good results with no tradeoff between recall and precision as can be observed in [9]. There are also algorithms that are capable to automatically detect duplicated functions by extracting metrics from the source code [10].

Abstract syntax tree clone analysis is more accurate than a line by line analysis or programming language token based approach due to the fact that it builds the abstract syntax tree (AST) [11]. The matching algorithm is based on hashing of each subtree in the AST and then placing each subsequence of the same length in similar buckets (groups) based on the similarity of the hash. All of the subtrees and subsequences in a bucket are compared against the similarity threshold. Thus, the algorithm requires that clones match by exceeding the similarity threshold at each particular AST node in the hierarchy. The fact that the algorithm rely on building all of the AST subtrees and making relatively costly operations, has an impact on algorithm's performance. The algorithm runs in *O(|Subtrees of AST|)*. On the other side, manipulation of the AST allows the tool to correctly detect even statement reordering and statement insertions.

The proposed clone detection solution is based on the algorithm presented in [2], with improvements regarding the way in which the sequence clones are detected and the way in which results are presented. Clone detection tool presented in [2] offers more detailed statistics regarding clones and also suggests possible refactorization, while the proposed solution focuses on clone detection, and reduces the running time of the algorithm by making a lot of parallel computations, very useful in the application of detecting student plagiarism over sets of assignments files. The clone detection algorithm has the following steps:

- parse the source code and creating the abstract syntax trees,
- hash each subtree of the abstract syntax trees and group them in different buckets based on their hash value,
- apply the clone detection algorithm, which has three sub-algorithms: *basic algorithm* – used to detect sub-tree clones; *sequence detection algorithm* – detection of statement and declaration sequence clones; *generalization algorithm* – search for more complex near-miss clones in order to generalize combinations of other clones,
- statistical results.

The **basic algorithm** finds subtree clones which in theory is an easy task: compare every subtree to every other subtree for equality. But, practically several issues appear:

- detection of near-miss clones
- sub-clone problem
- scaling

Near-miss clones are obtained after performing small adaptations on copied code fragments and they can be detected by comparing subtrees for similarity instead of equality. The sub-clone problem refers to reporting only maximally large clones and ignoring the others. The scaling problem represents the main drawback of this algorithm due to the high number of subtrees that need to be compared. Comparing every subtree to every other subtree takes too much time for large software systems. As stated in [2] for a software system having M lines of code there are *N=10*M AST* nodes and the total time for subtree comparison is *O(N^3)*. A solution to this problem is to use a hash function to hash the subtrees and based on the hash value to place each subtree in a bucket. Detection of the exact subtree clones is straightforward as only the subtrees in the same bucket have to be compared to each other. In this way, the number of comparisons is reduced by a factor proportional with the number of buckets; this step occurs in *O(N)* time.

Comparing for exact equality finds only exact clones and excludes the possibility to find near-miss clones. For detecting near-miss clones it is necessary to use a hash function that ignores small subtrees and to compare subtrees for similarity instead of exact equality defining a similarity threshold parameter. The similarity threshold parameter let the user choose how similar two subtrees should be. When comparing subtrees to each other if their similarity is greater than the similarity threshold parameter then they are considered clones. Similarity of two subtrees is computed using the following formula: $2 * S / (2 * S + L + R)$, where S represents the number of shared nodes, L and R represents the number of nodes that are part of only one of the two subtrees.

The **sequence detection** algorithm is capable of detecting statement sequence clones in abstract syntax trees using the basic algorithm. Such sequences appear in abstract syntax tree as right-leaning or left-leaning trees with some kind of identical sequencing operator as root. When the parser generator produces parsers that automatically generate abstract syntax tree, it is common that the trees have a left-leaning shape.

The sequences of the subtrees are not strictly trees and require a special treatment. Generic clone detectors are not able to detect tree nodes which constitute sequence nodes. Thus, these sequence nodes have to be explicitly identified to the clone detector tool. The basic algorithm doesn't detect them as a single subtree but rather as a sequence of subtrees and therefore it detects individual statements as clones but is not capable of detecting that these statements are in fact part of the same maximal clone. The proposed clone detection algorithm detects all the sequences of statements from the source code and keeps them in a list. These sequences are actually formed of subtrees of the abstract syntax tree. In our proposed solution, the sequence algorithm has two independent steps: in the first step it searches for clones by looking inside of each

sequence and in the second step it searches for clones in two different sequences. The sequence algorithm compares for equality the hash values of each subtree contained in the analyzed pair of sequences, detecting clones that are formed of at least two subtrees.

The **generalization algorithm** finds more complex near-miss clones. The proposed method is to visit the parent node of the already detected clones and verify if the parent node is a near-miss clone too. If the parent node is a near-miss clone, than the two clones are merged together. All the near-miss clones of this type are assembled from some set of exact sub-clones and therefore no near-miss clones will be missed. The set of detected clones will be marked and the clone pairs that are contained in other clone pairs will be removed. The detected set of clones is the union of all the clones found after the three algorithms have been applied.

Code clone analysis using abstract syntax trees presents high accuracy but because of intensive operations performed has the disadvantage of being much slower than other less accurate algorithms. We improved the algorithm's performance by using a parser with fewer ambiguities which will reduce the number of abstract syntax tree nodes and by parallelizing comparisons of the subtrees.

III. IMPLEMENTATION

The proposed algorithm makes use of the abstract syntax tree approach for detecting code clones. Several issues must be considered when building a clone detector tool using this approach: parsing and building the abstract syntax tree, handling preprocessor directives (when it is the case), applying code clone detection algorithms, reporting results and scaling the algorithm to industrial size source code. The diagram presented in the Fig. 1 represents an overview of the entire system.

The clone detection tool was designed to be extendable with new functionalities, including easy integration of support for other programming languages. Having a modular architecture increases the resistance to future changes. The main issue of the clone detector using the abstract syntax tree approach is that the clone detection algorithm takes much longer run-time than other algorithms. For this reason, we focused on the performance.

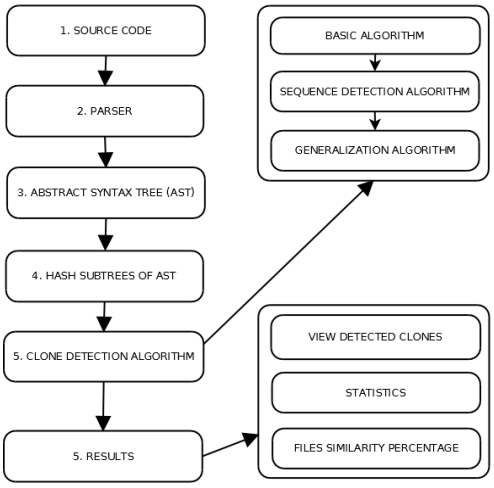

Figure 1. System architecture

Also, the high amount of data to be processed must be considered and therefore during the development it was made extensive use of the multi-threading. Concurrency was used whenever possible to do computations in parallel and the system was designed to support parallel computing. Because Java programming language was used to implement the tool, the task of scaling the application to the number of available CPUs was left in the care of the JVM (Java Virtual Machine).

A. Basic Algorithm

The basic algorithm is the first one of the three algorithms that altogether constitutes the clone detection algorithm. The input is represented by the subtrees of the abstract syntax tree which were hashed and grouped in buckets during the parsing process. The algorithm compares all the subtrees in the same bucket and detects the exact clones and near-miss clones where only the variable names were modified. Because during the parsing process variable names were replaced with the generic value "*<ID>*", the subtrees in the same bucket will be exactly the same, except for the cases where a hash collision occurres.

In the Fig. 2 it is presented the relation between the classes of the basic algorithm module. The *ComparisonManager* class is the entry point of the basic algorithm and starts the algorithm and stops it before it finishes. Subtree comparison is performed in a user configurable number of comparison threads. Taking into consideration that only subtrees from the same bucket will be compared with each other, it can be concluded that the work unit is represented by a bucket and that each comparison thread should get a number of buckets to process. The *LoadBalancer* class gets as input all the buckets and the number of comparison threads that will be used and assigns each bucket to a comparison thread. In order to distribute the buckets across comparison threads, the *LoadBalancer* computes a complexity coefficient for each bucket. Complexity coefficient results as multiplication of the number of subtrees in the bucket with the number of abstract syntax tree node of a single subtree.

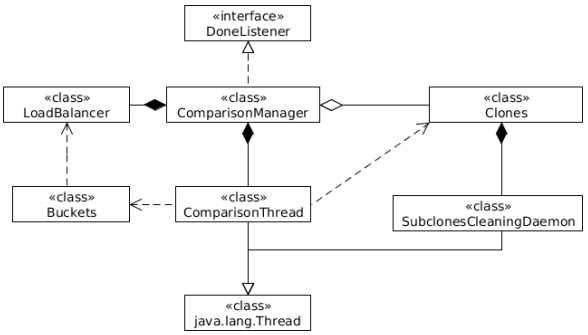

Figure 2. The Basic Algorithm

The *ComparisonThread* instances are managed by the *ComparisonManager* class, and started once the buckets have been assigned. Comparison threads will compare for similarity all the subtrees in the same bucket and a similarity coefficient is computed for each pair. If the

pair's similarity coefficient is greater than the similarity threshold (defined in the *ComparisonManager*) then the two subtrees form a clone.

The *Clones* class is used all over the clone detection algorithm to manage the detected clones. It stores all the clones in a map where the key is a subtree and the value is represented by a list of subtrees, each one being clones of the subtree key. Each of the subtrees in a pair that form a clone are stored twice, one being the key and the other one being in the list of clones for that key and vice versa. Also, it manages the *SubclonesCleaningDaemon* class, which is a thread that removes the clones that are not maximally large (further on referred as sub-clones).

B. Sequence Algorithm

The sequence algorithm is the part of the clone detection algorithm that handles the detection of code clones within sequences of statements or declarations. When parsing a source code file all the sequences from that file are identified and stored in the *SequenceList* class for future usage. Identification of sequences depends on the programming language in which the source code analyzed is written. In C programming language, sequences can be found inside functions, delimited by "{" and "}".

In Fig. 3 is depicted the relationship between classes that are part of the sequence algorithm module. As could be observed from the diagram, the sequence algorithm has two steps that run sequentially. The first step named subsequence algorithm, takes care of detecting code clones inside each sequence, and verifies if two subsequences of the same sequence are clones. The second step is named sequence algorithm and handles the detection of sequences that are clones.

The sequence algorithm also makes extensive use of multi-threading, the *SequenceManager* is responsible for starting and stopping the sequence algorithm. The *SequenceLoadBalancer* class balances the amount of work between worker threads for the both steps of the algorithm. Due to the fact that *SubsequenceThreads* and *SequenceThreads* compare subtrees using their hash value, renaming variables do not represent a problem because they were already replaced with a general placeholder when parsing the source file. False positives, as the result of comparing hash values, represent a known and accepted risk. Clones detected during the sequence algorithm are taken in consideration only if they are composed of at least two subtrees.

Sub*SequenceThreads* and *sequenceThreads* do not add the detected clones directly into the main clones list of *Clones* class, instead they add the clones into a separate list that will be merged with the main clone list at the end of sequence algorithm. Two internal classes of *SequenceManager*, named *SubsequenceAlgorithmDone* and *SequenceAlgorithmDone* implement the *DoneListener* interface and they are notified when one of the *SubsequenceThreads* or *SequenceThreads* finish its job. When all the *SubsequenceThreads* are finished the second step of the algorithm can start.

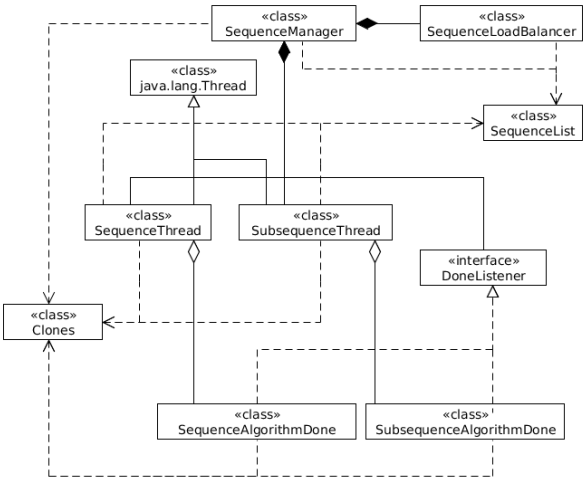

Figure 3. Sequence algorithm

When all the *SequenceThreads* are finished, the sequence algorithm is finished, the separate clone list is merged with the main clone list and *SubcloneCleaningDaemon* runs and removes all the new subclones.

C. Generalization Algorithm

The generalization algorithm is the last of the three algorithms that altogether form the clone detection algorithm. It takes care of detecting near miss clones obtained by inserting or removing statements after copying a code fragment or by performing other changes on the copied fragment, except renaming the variables. The generalization algorithm verifies if the parents of the already detected clones are also near-miss clones. The generalization algorithm compares subtrees for similarity using the similarity threshold parameter configurable at runtime. The similarity threshold parameter's value is important because most of subtrees have many nodes and a loose value can allow too many differences between subtrees and a tight value could prevent detecting some interesting clones.

Figure 4 contains the UML class diagram which describes the relations between classes that are involved in the generalization algorithm.

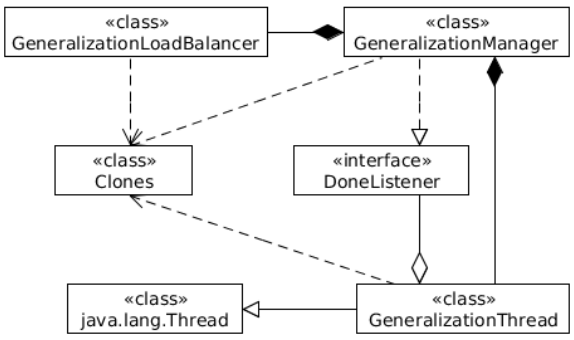

Figure 4. Generalization algorithm

The *GeneralizationManager* class is the main class of the generalization algorithm being responsible of starting and stopping the algorithm. The *GeneralizationLoadBalancer* computes the number of abstract syntax tree nodes and based on this value it assigns the subtree key from the main clone list to a *GeneralizationThread*. The *GeneralizationThread* applies the generalization algorithm to all the previously detected clones. Before reaching the comparison stage, the pair and each of the subtrees in the pair must meet a few conditions: each of them must have a parent node in the abstract syntax tree, and the parent nodes must be different. Also, because in the main clones map each clone is stored twice, the generalization algorithm will be run only once for each clone.

Clones detected during the generalization algorithm are not added directly into the main clones list, instead they are added into a separate clone list which will be later merged with the main list. Generalization algorithm consist of several iterations and when no clones where detected during one iteration the algorithm finishes.

IV. CASE STUDY

This chapter presents and analyses the results obtained after running the clone detection algorithm on multiple source codes in the C programming language.

The clone detection algorithm was evaluated by detecting plagiarism over student assignments in the Programming Techniques class. Each student implemented 5 assignments out of 10 available. The algorithm was run separately on the source files of each assignment and several metrics were computed: running time of the clone detection algorithm, number of detected clones, and percentage of code cloned. All the experiments were performed on a Linux machine with Intel Dual-Core 2 GHz CPU and 3 GB RAM.

Table I presents in detail for each of the 10 assignments: the sum of the number of lines of code, the number of detected clones, the percentage in which code was cloned in each case and the running time of the algorithm.

TABLE I
Aggregated clone detection metrics per each assignment

	Lines of code	Number of clones	Clone percentage (%)	Running time (s)
P1	13913	1087	63	138
P2	7732	453	57	16.1
P3	4048	187	38	4
P4	10217	497	66	11.3
P5	8346	455	43	4.2
P6	2436	185	46	1.7
P7	4499	345	62	6
P8	2856	117	47	1.7
P9	2623	108	59	2
P10	2738	89	42	1.8

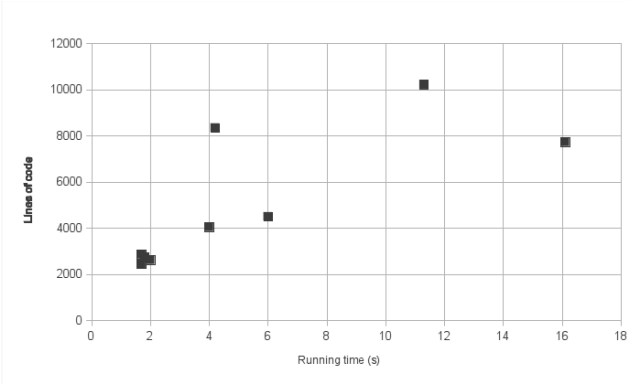

Figure 5. Running time related to the number of lines of code, P2 - P10

As can be observed from Fig 5, the running time generally increases with the size of the source code to analyze, but still there are exceptions, like in the case of P5 which has almost twice the size of P7, but clone detection algorithm run faster. The explanation for these situations regards the types of encountered clones detected during the three algorithms: basic, sequence and generalization algorithm.

In the Table II it is presented a statistic regarding the number of pairs of files having their similarity percentage of at least 40%, for each of the 10 assignments, over a sample of 100 source code files for each assignment. This statistic is useful in order to have an overview of the scale at which source code was duplicated. E.g. for assignment P1 there are 200 pair of source files having the similarity percentage between 50-60%. From the total amounts depicted in the Table II can be observed that the highest obtained values regarding the percentage of clone detection between the same assignment of different students were 399 (meaning that 399 source code files were detected as being clones in percentage between 90% and 100%) and 362 (meaning that 362 source code files were detected as being clones in percentage between 50% and 60%).

TABLE II
Percentage statistics per assignment

% / A	40-50 %	50-60 %	60-70 %	70-80 %	80-90 %	90-100 %
P1	6	200	47	50	8	155
P2	1	58	54	30	6	20
P3	0	3	0	0	0	16
P4	0	14	8	0	2	98
P5	1	10	5	13	10	6
P6	7	9	5	3	5	15
P7	1	66	15	1	4	28
P8	0	0	0	3	3	28
P9	2	0	6	4	0	17
P10	0	2	9	5	2	16
Σ	18	362	149	109	40	399

V. CONCLUSIONS

The clone detection method presented in the current paper offers the possibility to analyze programs written in the C programming language by detecting the number of clones and the percentage in which two source files are similar, based on the number of lines of code that are part of clones located in the two files. The tool can also be used to detect plagiarism, as was presented in the previous chapter, being very useful for teachers that are in the position of verifying students assignments.

It could be easily extended to support a large variety of programming languages just by implementing a new parser. Due to the fact that it is implemented using the Java programming language it runs on all major platforms: Linux, Mac OS X and Windows.

REFERENCES

[1] R. Wettel and R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments" in *Proc. 7th Int'l. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2005, pp. 63-70.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", in *ICSM*, November 1998, pp. 368-377.

[3] C. K. Roy and J. R. Cordy, *A survey on software clone detection research. Technical report*, Queen's University at Kingston, Ontario, Canada, 2007.

[4] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", in *ICSM*, 30 August – 3 September, 1999, pp. 109-118.

[5] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees", in *Proc. 13th Working Conference on Reverse Engineering*, October 2006, pp. 253-262.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.

[7] C. Kapser and M. Godfrey, *Improved tool support for the investigation of duplication in software*, 2005.

[8] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code", in *Proc. Int. Symposium on Static Analysis*, July 2001, pp. 40-56.

[9] J. Krinke, "Identifying similar code with program dependence graphs", in *Proceedings Eigth Working Conference on Reverse Engineering (WCRE'01), IEEE Computer Society*, pp. 301-309, October 2001.

[10] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *International Conference on Software System Using Metrics*, 1996, pp. 244-253.

[11] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. In Proc. *Spring Young Researchers Colloquium on Software Engineering*, May 2008, pp. 51-54.