LLVM-Based Code Clone Detection Framework*

Arutyun, Avetisyan
ISP RAS
Moscow, Russia
arut@ispras.ru

Shamil, Kurmangaleev
ISP RAS
Moscow, Russia
kursh@ispras.ru

Sevak, Sargsyan
ISP RAS
Moscow, Russia
sevaksargsyan@ispras.ru

Mariam, Arutunian
ISP RAS
Moscow, Russia
arutunian@ispras.ru

Andrey, Belevantsev
ISP RAS
Moscow, Russia
abel@ispras.ru

ABSTRACT
Existed methods of code clones detection have some restrictions. Textual and lexical approaches cannot detect strongly modified fragments of code. Syntactic and metrics based approaches detect strong modifications with low accuracy. On the contrary, semantic approach accurately detects the cloned fragments of code with small changes as well as the strongly modified ones. Methods based on this approach are not scalable for analysis of large projects. This paper describes LLVM-based code clone detection framework, which uses program semantic analysis. It has high accuracy and is scalable for analysis million lines of source code. The tool embeds a testing system, which allows generating code clones for the project automatically. It is used for determining the developed algorithms accuracy. The instrument is applicable for all languages that can be compiled to LLVM bitcode. Proposed method was compared with two widely used tools MOSS and CloneDR. Results show that it has higher accuracy. The tool is scalable for analysis of linux-2.6 kernel, which has about fourteen millions lines of source code.

Keywords
Code clone, program dependence graph, LLVM

1. INTRODUCTION
Software developers often reuse the same fragments of code many times by making small modifications. Hard deadlines usually increase copy-paste activities, which increase the number of code clones. Code cloning can lead to many semantic errors. For example, software developer can forget to rename some variable after copy-paste. The software, which has many clones, probably will have many mistakes and low quality. According to different studies [1, 2] up to 20% of source code can belong to clones. Clone detection tools are widely used:

- During software development to avoid mistakes and improve its quality;
- For automatic refactoring;
- For code size optimizations;
- For semantic errors detection.

The goal of this paper is to introduce LLVM-based code clone detection framework. It is based on semantic analysis of the program and is scalable up to millions lines of source code. The instrument consists of three basic parts.

The first part is responsible for program dependence graphs (PDG) generation. PDGs are constructed during project's build time, which allows creating these graphs without additional source code analysis.

The second part analyzes PDGs for code clones detection. It contains a number of new algorithms for PDGs' splitting and similar subgraphs detection. Due to the use of combined algorithms the tool is scalable up to millions lines of source code. Two types of algorithms are used for maximal isomorphic subgraphs detection. The first type of algorithms tries to prove that the pair of PDGs cannot have the desired isomorphic subgraphs. The most of PDGs' pairs are processed by them. These algorithms have liner complexity. The second type is approximate algorithms for maximal isomorphic subgraphs detection. These algorithms are applied if algorithms of the first type are failed. They have high computational complexity.

The third part is responsible for testing the developed algorithms. It automatically generates a set of code clones for a project and runs the clone detection algorithms. The number of clones detected by the specific algorithm specifies its correctness.

* The paper is supported by RFBR grant 15-07-07541.

2. BACKGROUND
2.1. Clone types
There are three basic clone types [3]. The first type is the identical code fragments except the variations in whitespace (may be also variations in layout) and comments (T1). The second type is the structurally/syntactically identical code fragments except the variations in identifiers, literals, types, layout and comments (T2). The third type is the copied fragments of code with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments (T3) (Fig. 1).

Original source
```
void sumProd(int n) {
  float sum = 0.0;
  float prod = 1.0;
  for (int i = 1; i<=n; i++) {
     sum = sum + i;
     prod = prod * i;
     foo(sum, prod);
  } }
```

Clone Type 1
```
void sumProd(int n) {
  float sum = 0.0; //C1
  float prod = 1.0; // C2
  for (int i = 1; i <= n; i++) {
  ____      sum = sum + i;
  ____      prod = prod * i;
  ____      foo(sum, prod);
  }}
```

Clone Type 2
```
void sumProd(int n) {
  int  s = 0; //C1
  int  p = 1; // C2
  for (int i = 1; i <= n; i++) {
  ____      s = s + i;
  ____      p = p * i;
  ____      foo(s, p);
  } }
```

Clone Type 3
```
void sumProd(int n) {
  int  s = 0; //C1
  int  p = 1; // C2
  for (int i = 1; i <= n; i++) {
  ____      s = s + i * i;
            // deleted
  ____      foo(s, p);
  } }
```

Figure 1. Examples for three clone types.

2.2. Code clone detection approaches
There are five [4, 5] basic approaches for code clone detection.

- Methods based on textual approach consider the source code as text and try to find equal substrings [6]. These substrings are clones. When all clones are found, clones which are located nearby can be combined to one. Basically (T1) clones are found.
- In case of lexical approach source code is parsed as a sequence of tokens. Then the longest common subsequence is determined. There are a few effective algorithms based on the parameterized suffix tree for clone detection [7]. One more interesting method transforms Java code to an intermediate representation and compares them instead of the original source [8]. These types of algorithms can find basically (T1) and (T2) clone types.

- The next is the syntactic approach. The algorithm works on Abstract Syntax Tree (AST). In this case clones are matched AST subtrees. Some algorithms directly compare two ASTs to find common subtrees [9]. Another algorithm constructs vectors of AST subtrees and compares them [10]. Algorithms based on this approach find all three types of clones.
- Metrics-based algorithms are widely used for clone detection. Algorithms based on this method compute a number of metrics for code fragments and compare them. Basically these metrics are computed for ASTs and PDGs [11]. Another method clusters computed metrics by using neural networks [12]. Metrics-based algorithms have better performance than AST or PDG comparison algorithms, but have low accuracy.
- The last one is the semantic approach. The source code is parsed to PDG. PDG nodes are program instructions whereas PDG edges are dependences between those instructions. Algorithms based on PDG try to find maximal isomorphic subgraphs for a pair of PDGs [13, 14, 15]. All algorithms are approximate because maximal isomorphic subgraphs detection is an NP-hard problem. PDG-based methods have high accuracy but low performance.

Textual and lexical approaches are not effective for detecting clones of (T3) type. AST and metrics based methods detect (T3) type of clones with low accuracy. Only semantic analysis allows reaching high accuracy.

3. PDG GENERATION

PDGs for the project are generated based on the LLVM intermediate representation called a bitcode. The LLVM pass is added for these graphs generation (see Fig. 2). The generation happens during the project compile time. It allows constructing graphs for large scale projects effectively. PDG graph's vertices are LLVM bitcode [16] instructions. Edges are obtained based on LLVM use-def [16], alias and control flow analyses. Those vertices which have no edges are removed, after which the optimized PDGs are stored to files. The tool allows generating PDG graphs in three different ways. Edges of the minimal PDG are constructed based only on LLVM use-def analysis. The middle level PDG also includes edges obtained by the alias analysis. The full PDG contains all data and control dependencies. This approach allows avoiding wasting unneeded resources, e.g., the minimal PDG is enough for accurate detection of T1 and T2 clone types. LLVM provides compiler APIs and has a large set of optimization libraries. Due to this, many programming languages provide source code translation to LLVM bitcode. Therefore we can apply the developed tool for all these languages.

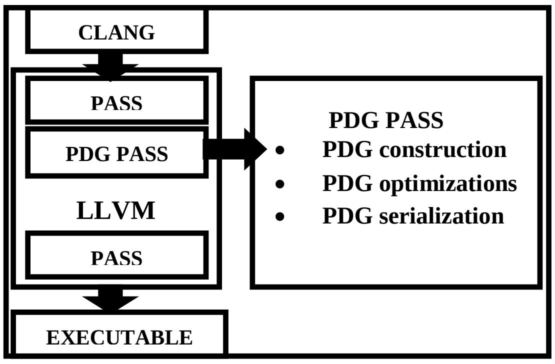


Figure 2. LLVM based model for PDGs' generation.

4. CLONE DETECTION

The clone detection is a multistage process. First, generated PDGs are loaded to memory, and then four basic steps are performed (see Fig. 3).

The first step is splitting PDGs to subgraphs. These subgraphs are considered as potential clones of each other. The second step is the application of fast check algorithms. These algorithms have linear complexity and try to prove that a pair of PDGs cannot have big enough isomorphic subgraphs. The third stage is the maximal isomorphic subgraphs detection. New algorithm, based on slice (see Section 4.3) is proposed for maximal isomorphic subgraphs detection. The fourth step is the filtration of the obtained pairs of maximal isomorphic subgraphs. The last step is printing of the corresponding source code for isomorphic subgraphs as detected clones.

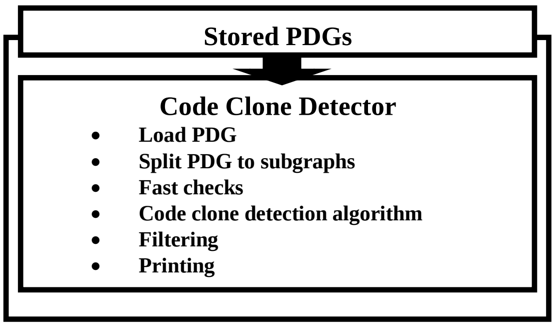
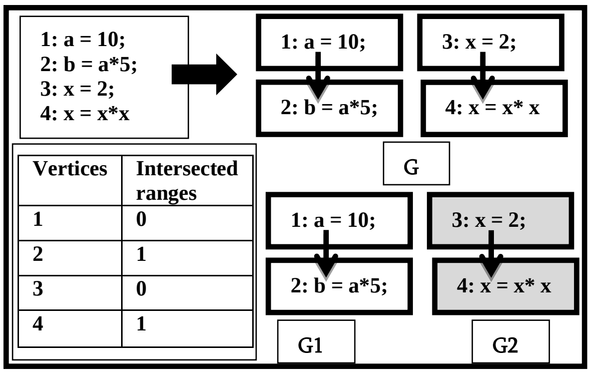


Figure 3. Basic stages of code clones detection.

4.1. Splitting PDGs

Three methods are implemented for splitting. The first method splits PDG to weakly connected components. The second method splits PDG to subgraphs, where every pair has less than *N* common nodes [17]. These two methods have two basic disadvantages: subgraphs' sizes might have big variation; corresponding source code lines for one subgraph might be located far from each other. To avoid these disadvantages, the third method is proposed. PDG edges are considered as source code ranges (see Fig. 4).



Vertices	Intersected ranges
1	0
2	1
3	0
4	1

Figure 4. Example of PDG's splitting.

For PDG's corresponding source code lines, the numbers of intersected ranges are considered. Source code is split based on those lines, which have minimum number of intersecting ranges. Corresponding subgraphs for split code fragments are considered for clone detection. Experimental results show that this splitting method allows detecting about 1.5-2 times more clones than the first and the second methods.

4.2. Fast checks

These algorithms have liner complexity and try to prove that the pair of PDGs does not have big enough isomorphic subgraphs. Two nodes of PDG are similar if their types are the same. Fast check algorithms compare PDG's nodes

based on their types. If the algorithm was not able to detect enough pairs of similar nodes in the corresponding graphs, these graphs cannot have big enough isomorphic subgraphs.
The first algorithm stores PDG nodes in a hash set, the key for the set is the node's type. If the size of intersection, for the sets of corresponding pair of PDGs, is not big enough then this pair of PDGs does not have the desired isomorphic subgraphs.
The second algorithm computes a characteristic vector for every PDG. Elements of this vector are count of nodes with specific type. If the Euclidean distance for corresponding vectors of considered pair of PDGs is too big then this pair fails the fast check.

4.3. Slice-based clone detection

For the given PDG's pair, candidate pairs of nodes are constructed. The first node in the pair is from the first PDG, the second one from the second PDG. For every pair of nodes backward and forward slices [13] are applied to construct isomorphic subgraphs. Maximal isomorphic subgraphs are selected from the constructed set of isomorphic subgraph pairs.
Two approaches are developed for candidate set construction. The first approach chooses for every node of the first PDG the most similar node from the second PDG. Metrics [18] are used for similar nodes detection. For all nodes of both PDGs bit vectors [18] are constructed. The most similar nodes are chosen based on similarity function [18]. The second approach considers vertices with a maximal number of neighbors from the first PDG and tries to find identical vertices (with neighbors) from the second PDG.

4.4. Filtration

The last stage in the process of code clone detection is the filtration of some detected pairs of isomorphic subgraphs. The need for a filter arises from the fact that the concept of code clone is defined for source code of the program, but isomorphic subgraphs are considered as clones. A code clone must present a sequence of lines in the file (not necessarily consecutive, but not highly dispersed). The purpose of filtering is to verify that the source code for the corresponding isomorphic subgraphs is not much scattered.

5. AUTOMATIC CLONE GENERATION

Two approaches are suggested for automatic generation of code clones. The first method uses obfuscation [19] and standard transformation, optimization passes of LLVM. For every function of LLVM bitcode two PDGs are constructed. The first PDG comes from the original code and is constructed based on LLVM bitcode generated by the Clang compiler. The second PDG is the clone PDG, and it is constructed based on the transformed/obfuscated bitcode. Standard passes of LLVM are applied to bitcode for transformation (see Fig. 6).

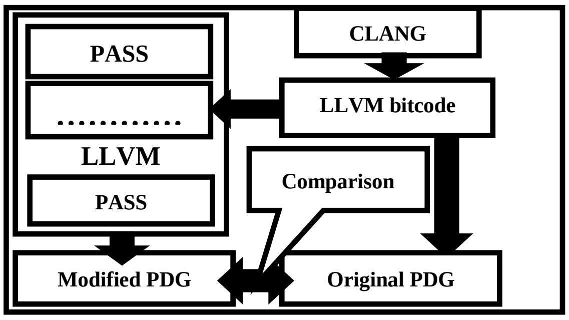


Figure 6. LLVM-based clone generation model.

The second method merges the original program PDGs to generate code clones (see Fig. 7).

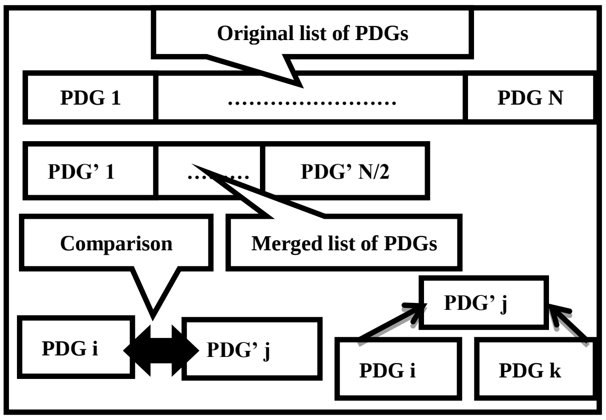


Figure 7. Clone generation based on PDG's merging.

Three methods are applied for PDGs' merge. The first method performs the union of two PDGs without adding extra edges or vertices. The second method unions a pair of PDGs and also adds extra random edges between the nodes of the corresponding graphs. The third method considers nodes of the first PDG and tries to find the similar nodes in the second PDG. If the similar node is detected then all neighbors of this node are added to the first PDG with their corresponding edges.
To check correctness of the implemented clone detection algorithms, original and cloned PDGs are compared. The number of detected clones specifies the correctness and power of the tested algorithm.

6. RESULTS

The developed tool was applied to a number of widely used libraries and software systems. It was compared with other tools of clone detection. The tests were run on a machine with Intel Core i3 CPU 540 and 8GB RAM.

6.1. Comparison with other tools

The described methods were compared with two widely used tools. The first one is MOSS [20]. It has been developed for detecting plagiarism in programming classes (Stanford University). The second one is CloneDR [21]. It was developed by Semantic Designs Company, which provides different tools for software design and analyses. The test suite is described in Table 1. The first test (Original Code) was modified in different ways to obtain all three types of clones. The paper [22] contains more details for all tests. Theoretically all files are clones, because they were obtained by modification of the single test. Clone detection tool with high accuracy should determine as much clones as possible. Tab.1 shows results of comparison for MOSS, CloneDR and developed three methods for clone detection.

Test Name	MOSS	CloneDR	Our tool
copy00.cpp	yes	yes	yes
copy01.cpp	yes	yes	yes
copy02.cpp	yes	yes	yes
copy03.cpp	yes	yes	yes
copy04.cpp	yes	yes	yes
copy05.cpp	yes	yes	yes
copy06.cpp	no	yes	yes
copy07.cpp	yes	yes	yes
copy08.cpp	no	no	yes

copy09.cpp	no	yes	yes
copy10.cpp	no	yes	yes
copy11.cpp	no	no	yes
copy12.cpp	yes	yes	yes
copy13.cpp	yes	yes	yes
copy14.cpp	yes	yes	yes
copy15.cpp	yes	yes	yes

Table 2. The results of comparison: "yes" - clone is found, "no" - clone is not found.

6.2. PDG generation time

Fig. 8 shows lines of source code for analyzed projects. Linux kernel has about fourteen million lines of source code written in the C language. Fig. 9 shows compilation time of the project with and without PDGs' generation. In the worst case compile time increases only by ~30%. Fig. 10 shows sizes of generated PDGs' for all projects.

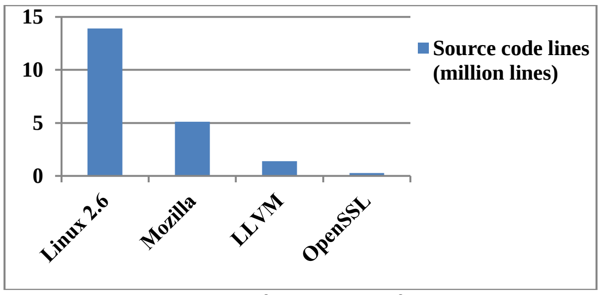


Figure 8. Lines of source code for projects

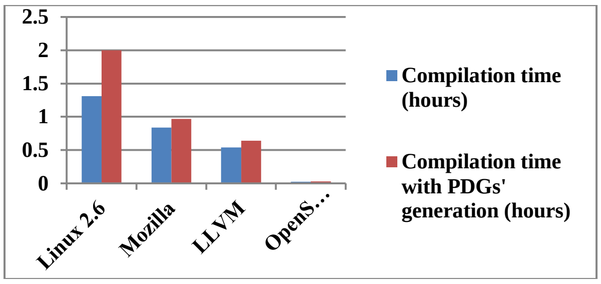


Figure 9. Comparison of compilation time for projects.

6.3. Detected clones

Fig. 10 shows the number of detected clones and the false positive rates. The Linux kernel contains about 2000 clones. The manual analysis identifies only 73 false positive clones.

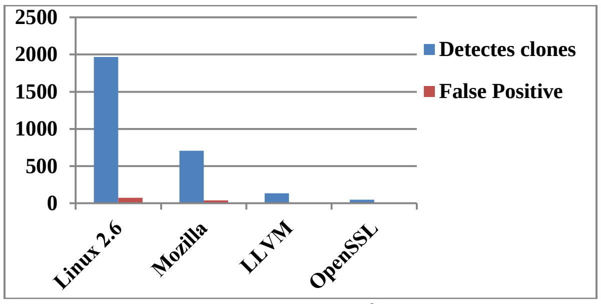


Figure 10. False positive rate of clone detection.

7. ACKNOWLEDGEMENT

The paper is supported by RFBR grant 15-07-07541.

REFERENCES

[1] B. Baker, "On finding duplication and near-duplication in large software systems", *in: Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 86-95, 1995.

[2] C.K. Roy, J.R. Cordy, "An empirical study of function clones in open source software systems", *in: Proceedings of the 15th Working Conference on Reverse Engineering*, pp. 81-90, 2008.

[3]. S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and evaluation of clone detection tools", *Transactions on Software Engineering*, pp. 577–591, 2007.

[4]. D. Rattana, R. Bhatiab and M. Singhc, "Software clone detection", *Information and Software Technology*, vol. 55, no. 7, pp. 1165-1199, 2013.

[5]. S. Gupta and P. C. Gupta, "Literature Survey of Clone Detection Techniques", *International Journal of Computer Applications*, vol. 99, no. 3, pp. 41-44, 2014.

[6] S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code", *in: Proceedings of the 15th International Conference on Software Maintenance*, pp. 109-119, 1999.

[7] T. Kamiya, S. Kusumoto, K .Inoue, CCFinder: "A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, pp. 654-670, 2002.

[8]. R. Kaur, S. Singh, "Clone detection in software source code using operational similarity of statements", *ACM SIGSOFT Software Engineering Notes*, pp. 1-5, 2014.

[9] I. Baxter, A. Yahin, L. Moura, M. Anna, "Clone detection using abstract syntax trees", *in: Proceedings of the 14th IEEE International Conference on Software Maintenance*, pp. 368-377, 1998.

[10] L. Jiang, G. Misherghi, Z.Su, S.Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones", *in: Proceedings of the 29th International Conference on Software Engineering*, pp. 96-105, 2007.

[11] J. Mayrand, C. Leblanc, E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *in: Proceedings of the 12th International Conference on Software Maintenance*, pp. 244-253, 1996.

[12] N. Davey, P. Barson, S. Field, R. Frank, "The development of a software clone detector", *International Journal of Applied Software Technology*, pp. 219-236, 1995.

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

[14] J. Krinke, "Identifying similar code with program dependence graphs", *in: Proceedings of the 8th Working Conference on Reverse Engineering*, pp.301-309, 2001.

[15]. S. Sargsyan, S. Kurmnagaleev, A. Belevantsev, H. Aslanyan, A. Baloian, "Scalable code clone detection tool based on semantic analysis", *The Proceedings of ISP RAS*, pp. 39-49 2015.

[16]. http://www.llvm.org

[17]. M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones", *in: Proceedings of the 30th International Conference on Software Engineering*, pp.321–330, 2008.

[18]. S.S. Sargsyan, S.F. Kurmangaleev, A.V. Baloian, H.K. Aslanyan, "Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph", *Mathematical Problems of Computer Science*, pp. 54-62, 2014.

[19]. S.F. Kurmangaleev, V.P. Korchagin, V.V. Savchenko S.S. Sargsyan, "Building an obfuscation compiler based on LLVM infrastructure", *The Proceedings of ISP RAS*, pp. 77-92, 2012.

[20] http://theory.stanford.edu/~aiken/moss/

[21] http://www.semdesigns.com/products/clone/

[22]. Chanchal K. Roya, James R. Cordya, Rainer Koschkeb, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", *Science of Computer Programming*, pp. 470-495, 2009.