Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода

Севак Саргсян <sevaksargsyan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация: В статье предлагается новый метод поиска семантических ошибок, возникающих при неправильном копировании исходного кода в процессе разработки ПО. Метод состоит из двух основных этапов. На первом этапе производится поиск клонов кода на основе лексического анализа программы. Найденные идентичные последовательности лексем фильтруются путем частичного разбора. После чего в них остаются целостные конструкции, допускаемые языком программирования. На втором этапе производится анализ найденных клонов с целью обнаружения допущенных ошибок при копировании. Для этого строится и анализируется граф зависимостей программы (Program Dependence Graph - PDG). Предложенный подход реализован в компиляторной инфраструктуре LLVM/Clang, что позволяет эффективным образом производить анализ, во время компиляции проекта. Найденные ошибки выдаются в виде предупреждений для разработчика. В статье приводится результаты анализа ядра Linux 2.6 и Android 4.3. Инструмент обеспечивает точность выше 65%.
Ключевые слова: семантический анализ, семантические ошибки, поиск клонов, PDG, LLVM

DOI: 10.15514/ISPRAS-2015-27(2)-6

Для цитирования: Саргсян Севак. Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 93-104. DOI: 10.15514/ISPRAS-2015-27(2)-6.

1. Введение

В процессе разработки программного обеспечения (ПО) часто прибегают к копированию ранее написанного кода, что может стать причиной возникновения семантических ошибок в программе. Чаще всего ошибки возникают из-за необновленных имен переменных в скопированном участке. Программы, насыщенные клонами кода, с большой вероятностью будут

содержать много ошибок. Инструменты поиска клонов кода и семантических ошибок широко применяются в процессе разработки ПО. Согласно исследованиям [1, 2] до двадцати процентов кода может быть клонами. Существуют пять основных подходов к поиску клонов. Текстовый [3] и лексический [4] подходы не могут найти все 3 типа (раздел 2) клонов. Синтаксический подход [5, 6] и метод, основанный на метриках [7, 8] находят третий тип клонов с низкой точностью. Подход, основанный на семантическом анализе программы [9, 10, 11, 12], находит все три типа клонов кода с большой точностью, но у этого подхода большая вычислительная сложность. Есть две основные причины медленной работы: первая это повторный анализ исходного кода для построения PDG, вторая это поиск клонов кода на основе изоморфизма подграфов PDG. Известно, что поиск максимально изоморфных подграфов – NP-сложная задача, и для ее решения применяются приближенные алгоритмы, у которых сложность может быть кубической от количества вершин в PDG.

Анализ больших проектов с открытыми исходными кодами показал, что большое количество ошибок возникает из-за неверно адаптированного кода, так, например, репозитории FreeBSD и Linux по данным на 2013 содержали более 113 и 182 исправлений подобных ошибок [13]. Для поиска ошибок, возникающих из-за неправильной адаптации скопированного кода, как правило, используют методы, основанные либо на лексическом анализе, либо на синтаксическом анализе.

Методы, основанные на лексическом подходе, трансформируют исходный код в последовательность лексем и ищут идентичные последовательности лексем (клоны кода) [14]. На втором этапе проверяются переменные клонированных фрагментов кода, которые используются больше одного раза в данном фрагменте. Если соответствующие переменные во втором фрагменте имеют разные имена, тогда произошло некорректное переименование переменных и данный фрагмент содержит семантическую ошибку. Недостаток такого подхода заключается в большом количестве ложных срабатываний, так как подход не учитывает контекст участка программы вокруг потенциального клона.

Методы, основанные на синтаксическом подходе [13, 15, 16], трансформируют исходный код в абстрактное синтаксическое дерево (AST – abstract syntax tree), и ищут схожие поддеревья. Затем производится анализ найденных поддеревьев для выявления возможных ошибок. Большинство известных инструментов используют [13, 17] репозиторий программы, что является дополнительным ограничением. Каждое изменение в репозитории анализируется, рассматривается его влияние на AST каждой функции и производится поиск дефектов. Недостаток такого подхода заключается в том, что некорректно переименованные переменные влияют на вид AST, поскольку могут появляться/исчезать узлы дерева для выражений (рис. 1).

```
фрагмент 1:                фрагмент 2:
1. a = 1;                  1. c = 1;
2. b = 2;                  2. m = 2;
3. x = a * b;              3. y = a * m; // разработчик забыл заменить "a" на "c".

AST фрагмента 1:           AST фрагмента 2:

        =                          =                  =
       / \                        / \                / \
      x   *                      y   *              c   1
         / \                        / \
        =   =                      a   =
       / \ / \                        / \
      a  1 b  2                      m   2
```

Рис.1. Влияние имен переменных на вид AST, после некорректного переименования

Данная работа описывает новый подход нахождения семантических ошибок, возникающих при неправильной адаптации копированного кода. Он состоит из двух основных этапов. На первом этапе обнаруживаются все клоны второго типа (раздел 2) путем лексического анализа функции. Второй этап строит PDG для этой функции, чтобы найти ошибки, допущенные при копировании.

2. Типы клонов

Есть три основных типа клонов (классификация приведена из [18]). Первый тип (**T1**) – это те фрагменты кода, которые отличаются только пробелами и комментариями. Второй тип (**T2**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных и значениями переменных. Третий тип (**T3**) – это те фрагменты кода, которые отличаются пробелами, комментариями, именами переменных, типами переменных, значениями переменных. В конкретном фрагменте могут быть также добавлены или удалены некоторые строки.

3. Модель инструмента поиска семантических ошибок

Генерация лексем и построение PDG проекта производится во время компиляции проекта (рис. 2). Новый проход LLVM [19] строит последовательность лексем на основе промежуточного представления. Для каждой функции производится поиск клонов. Клонами считаются совпадающие последовательности лексем. Существуют ряд широко известных инструментов, которые работают на основе лексического подхода [4, 14]. Есть две основные причины того, что предлагаемый метод на первом этапе использует лексический анализ для поиска клонов кода. Основная причина возникновения семантических ошибок - это непереименование переменных после копирования участка кода (рис. 2). Лексический анализ не чувствителен к именам переменных, что помогает найти скопированные участки кода, в которых есть непереименованные переменные (семантические ошибки). При

использовании AST и PDG, имена переменных могут повлиять на вид графа (рис. 1, 5).

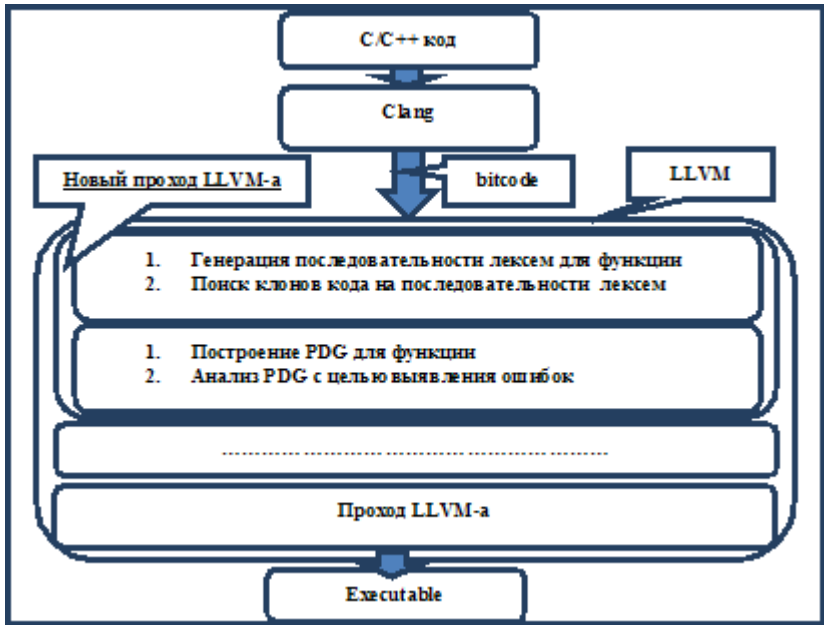

Рис. 2. Поиск клонов кода на базе компиляторной инфраструктуры LLVM.

Для функции, в которой найдена пара клонов, строится PDG для проверки корректности копирования. Вершинами PDG являются инструкции промежуточного представления LLVM. Ребрам соответствуют зависимости по управлению между инструкциями, зависимости, получаемые анализом алиасов и LLVM use-def анализом [19]. Такой подход имеет ряд преимуществ. Он позволяет без повторного анализа исходного кода получить лексемы и графы для больших проектов, содержащих миллионы строк исходного кода. Не требуется проводить дополнительный анализ зависимостей между единицами компиляции.

По сравнению с существующими методами [14, 15] предложенный подход обладает большей точностью, благодаря тому, что использует представление PDG, содержащее всю необходимую информацию о программе, для поиска ошибок.

4. Поиск клонов кода на основе лексического анализа

На первом этапе на основе промежуточного представления LLVM получается последовательность лексем функции (рис. 3). Предлагаемый алгоритм, обрабатывает последовательность лексем и находит все непересекающиеся

пары идентичных подпоследовательностей максимального размера. После этого производится частичный разбор идентичных последовательностей, для корректного определения синтаксических конструкций языка. Идентичные последовательности фильтруются от неполных конструкций (например, если с телом цикла вошли в подпоследовательность и другие лексемы, которые не представляют собой целую инструкцию или блок, то они удаляются из обеих последовательностей). Если отфильтрованные идентичные последовательности имеют достаточно большой размер, они передаются на второй этап для проверки.

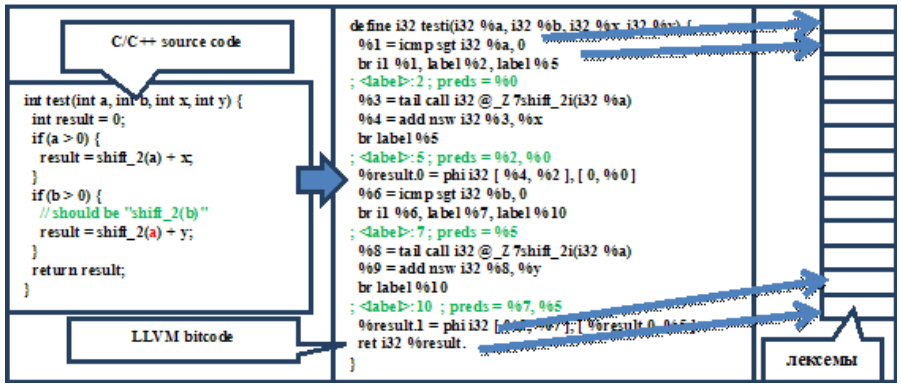

Рис. 3. Построение последовательности лексем.

5. Поиск ошибок

Для поиска ошибок в скопированном фрагменте кода, строится PDG граф соответствующей функции. В полученном графе выделяются два подграфа, соответствующие идентичным последовательностям лексем. Полученные подграфы расширяются путем добавления вершин (назовем эти вершин исходными вершинами - ИВ), соответствующих переменным, которые используются в выделенных подграфах (рис. 4).

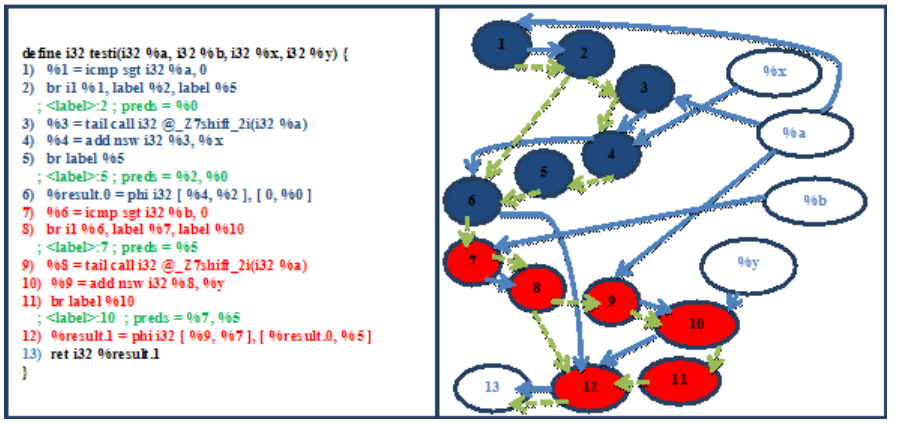

Рис. 4. PDG для функции. {1, 2, 3, 4, 5, 6, %x, %a}, {7, 8, 9, 10, 11, 12, %a, %b, %y} выделенные подграфы.

Если выделенные подграфы не изоморфны, то при копировании, с большой вероятностью, произошла ошибка, поэтому нужен дополнительный анализ. Анализ ИВ дает информацию о возможной ошибке. Если есть ИВ, которые входят в выделенные подграфы, и в каждом подграфе имеют разную степень (рис. 5), тогда переменные, соответствующие этим вершинам, содержат ошибочное использование.

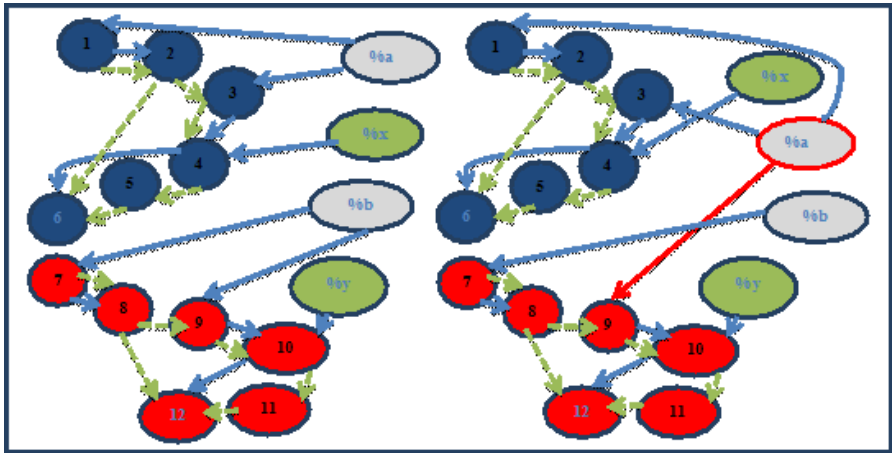

Рис. 5. Изменение структуры PDG при неправильном копировании кода.

6. Проверка изоморфизма на основе метрик

Для проверки изоморфизма пары PDG, сравниваются значения метрик, рассчитанные для ребер соответствующих графов. Каждая вершина PDG имеет индекс, который представляет собой код операции соответствующей инструкции промежуточного представления LLVM. Метрика ребра представляет собой число, которое получается на основе индексов соседних вершин и степени одной вершины.

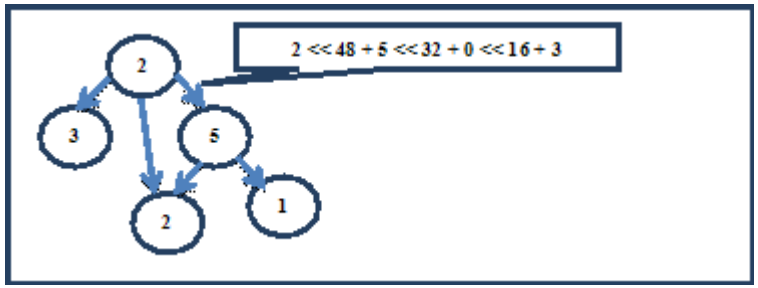

Рис. 6. Метрика ребер PDG

Из рис. 6 видно, что метрика ребра представляется в виде 64 битного целого числа. Первые 16 битов - это индекс первой вершины, следующие 16 битов - это индекс второй вершины, после этого следует количество входящих и выходящих ребер начальной вершины, которым также выделено по 16 битов. Алгоритм проверки изоморфизма вычисляет и сохраняет метрики ребер каждого графа в отдельном множестве, после чего проверяет равенство полученных множеств. Если они неравны, тогда пара PDG графов не может быть изоморфной.

7. Результаты

Ниже представлены результаты анализа нескольких проектов с открытым исходным кодом (рис. 7). Приведен пример семантической ошибки: разработчик не переименовал переменную 'userName' на 'password' (строка 277) после копирования (рис. 8).

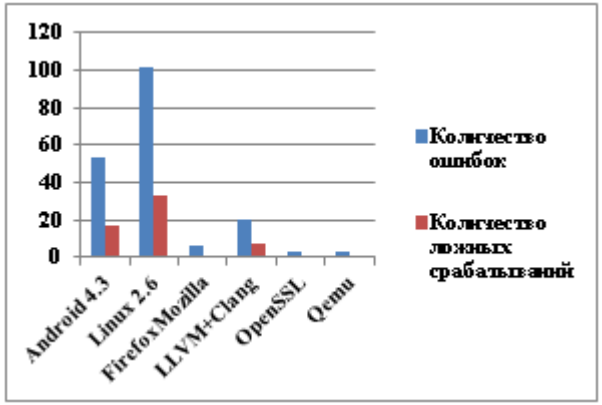

Рис. 7. График найденных семантических ошибок и ложных срабатываний.

Android 4.3 - AccountSetupBasics.java

```
271|    String userName = SetupData.getUsername();
272|    if (userName != null) {
273|        mEmailView.setText(userName);
274|        SetupData.setUsername(null);
275|    }
-------------------------------------------------
276|    String password = SetupData.getPassword();
277|    if (userName != null) {
278|        mPasswordView.setText(password);
279|        SetupData.setPassword(null);
280|    }
```

Рис. 8. Пример найденной семантической ошибки.

8. Заключение

В статье предлагается новый подход поиска семантических ошибок, возникающих при неправильном копировании участков исходного кода. Применяется гибридный анализ исходного кода для выявления возможных ошибок. Метод использует лексический анализ и частичный разбор для поиска клонов кода, после чего производится семантический анализ. В ходе семантического анализа выявляются некорректно обновленные участки кода, использование которых может содержать ошибку. Предложенный подход реализован в компиляторной инфраструктуре LLVM/Clang, что позволяет выявить семантические ошибки проекта во время компиляции.

Список литературы.

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

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

[3]. S. Ducasse, M. Rieger and S. Demeyer, A language independent approach for detecting duplicated code, Proceedings of the 15th International Conference on Software Maintenance, (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[4]. T.Kamiya, S.Kusumoto and K.Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, 2002.

[5]. I. Baxter, A. Yahin, L. Moura and M. Anna, Clone detection using abstract syntax trees, Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, pp. 368-377, 1998.

[6]. L.Jiang, G.Misherghi, Z.Su and S.Glondu, DECKARD : Scalable and accurate tree-based detection of code clones", Proceedings of the 29th International Conference on Software Engineering, (ICSE07), IEEE Computer Society, pp. 96-105, 2007.

[7]. J. Mayrand, C. Leblanc and E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, Proceedings of the 12th International Conference on Software Maintenance, (ICSM96), Monterey, CA, USA, pp. 244-253, 1996.

[8]. Sargsyan S., Kurmangaleev S., Baloian A., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, pp. 54-62, 2014.

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

[10]. J. Krinke, Identifying similar code with program dependence graphs, Proceedings of the 8th Working Conference on Reverse Engineering, (WCRE 2001), pp. 301-309, 2001.

[11]. Y. Higo and S. Kusumoto, Code clone detection on specialized PDGs with heuristics, Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR11), Oldenburg, Germany, pp.75-84, 2011.

[12]. С. Саргсян, Ш. Курмангалеев, А. Белеванцев, А. Асланян, А. Балоян, Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ, Труды Института системного программирования РАН Том 27. Выпуск 1. 2015 г.

[13]. K. Miryung, S. Person, N. Rungta, Detecting and characterizing semantic inconsistencies in ported code, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference, pp. 367-377

[14]. Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176-192.

[15]. P. Jablonski, D. Hou, CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE, in Proceedings of the 2007 OOPSLA workshop on eclipse technology, 2007, pp.16-20.

[16]. L. Jiang, Z. Su, E. Chiu, Context-Based Detection of Clone-Related Bugs, in Proceedings of the 6th joint meeting of the European software engineering conference, 2007, pp. 55-64.

[17]. Y. Higo, S. Kusumoto, MPAnalyzer: A Tool for Finding Unintended Inconsistencies in Program Source Code, in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp.843-846.

[18]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33 (9) (2007) 577–591

[19]. http://llvm.org

Copy-Paste Semantic Errors Detection

Sevak Sargsyan <sevaksargsyan@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004

Annotation. The paper describes a method for semantic errors detection arising during incorrect code copy-paste made by the developer. The method consists of two basic parts. The first part detects code clones based on lexical analysis of the program. A sequence of tokens is constructed based on the LLVM lexer and then all pairs of maximal, non-intersected matched token sequences are detected. The pairs of identical subsequences are then partially parsed to retain the constructs allowed by the programming language and to remove the incomplete sequences. When the remaining subsequences are big enough, the second stage is applied for them. A Program Dependence Graph (PDG) is constructed for the corresponding function code, and then identical subsequences' subgraphs are considered. If two subgraphs have shared vertices, then outgoing edges of these vertices are analyzed. This allows detecting semantic errors with high accuracy. The described method is implemented for the LLVM/Clang compiler. Due to this semantic mistakes are detected during program compile time, so there is no need for separate lexical and semantic program analysis. A number of widely used open source libraries and software systems were analyzed. The paper contains the list of detected semantic errors for Linux kernel 2.6 and Android 4.3. For these systems, the true positive rate achieved by our approach is above 65%.

Keywords: lexical analysis; semantic analysis; code clones; PDG; LLVM

DOI: 10.15514/ISPRAS-2015-27(2)-6

For citation: Sargsyan Sevak. Copy-Paste Semantic Errors Detection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 93-104 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-6.

References.

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

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

[3]. S. Ducasse, M. Rieger and S. Demeyer, A language independent approach for detecting duplicated code, Proceedings of the 15th International Conference on Software Maintenance, (ICSM'99), Oxford, England, UK, pp. 109-119, 1999.

[4]. T.Kamiya, S.Kusumoto and K.Inoue, CCFinder: A multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654-670, 2002.

[5]. I. Baxter, A. Yahin, L. Moura and M. Anna, Clone detection using abstract syntax trees, Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, pp. 368-377, 1998.

[6]. L.Jiang, G.Misherghi, Z.Su and S.Glondu, DECKARD : Scalable and accurate tree-based detection of code clones", Proceedings of the 29th International Conference on Software Engineering, (ICSE07), IEEE Computer Society, pp. 96-105, 2007.

[7]. J. Mayrand, C. Leblanc and E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, Proceedings of the 12th International Conference on Software Maintenance, (ICSM96), Monterey, CA, USA, pp. 244-253, 1996.

[8]. Sargsyan S., Kurmangaleev S., Baloian A., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, pp. 54-62, 2014.

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

[10]. J. Krinke, Identifying similar code with program dependence graphs, Proceedings of the 8th Working Conference on Reverse Engineering, (WCRE 2001), pp. 301-309, 2001.

[11]. Y. Higo and S. Kusumoto, Code clone detection on specialized PDGs with heuristics, Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR11), Oldenburg, Germany, pp.75-84, 2011.

[12]. S. Sargsyan, S. Kurmnagaleev, A. Belevantsev, H. Aslanyan, A. Baloian, Scalable code clone detection tool based on semantic analysis, The Proceedings of ISP RAS, vol. 27, issue 1, 2015.

[13]. K. Miryung, S. Person, N. Rungta, Detecting and characterizing semantic inconsistencies in ported code, Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference, pp. 367-377

[14]. Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering 32 (3) (2006) 176-192.

[15]. P. Jablonski, D. Hou, CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE, in Proceedings of the 2007 OOPSLA workshop on eclipse technology, 2007, pp.16-20.

[16]. L. Jiang, Z. Su, E. Chiu, Context-Based Detection of Clone-Related Bugs, in Proceedings of the 6th joint meeting of the European software engineering conference, 2007, pp. 55-64.

[17]. Y. Higo, S. Kusumoto, MPAnalyzer: A Tool for Finding Unintended Inconsistencies in Program Source Code, in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp.843-846.

[18]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33 (9) (2007) 577–591

[19]. http://llvm.org