Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ*

Севак Саргсян, <sevaksargsyan@ispras.ru>
Шамиль Курмангалеев, <kursh@ispras.ru>
Андрей Белеванцев, <abel@ispras.ru>
Айк Асланян, <hayk@ispras.ru>
Артем Балоян, <artyom@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация: В статье обсуждаются существующие методы поиска семантически сходных участков кода (клонов). Анализируются недостатки каждого метода, на основе чего предлагается новый метод поиска клонов кода и описывается архитектура инструмента для языков C/C++ на основе компиляторной инфраструктуры LLVM, в которой реализован предложенный метод. Работу инструмента можно разделить на два основных этапа. На первом этапе программа компилируется в промежуточное представление LLVM компилятором Clang. По этому представлению строится граф зависимостей программы (Program Dependence Graph – PDG) для каждой единицы компиляции. На втором этапе производится анализ поиска клонов кода в построенных графах. В инструменте существует отдельный этап тестирования алгоритмов, который будет подключен при запуске инструмента в режиме тестирования. Это дает возможность автоматической генерации тестов и проверки точности реализованных алгоритмов.

Ключевые слова: семантический анализ, поиск клонов, PDG, LLVM.

DOI: 10.15514/ISPRAS-2015-27(1)-3

Для цитирования: Саргсян Севак, Курмангалеев Шамиль, Белеванцев Андрей, Асланян Айк, Балоян Артем. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 39-50. DOI: 10.15514/ISPRAS-2015-27(1)-3.

1. Введение

Повторное использование фрагментов исходного кода часто встречается при разработке программного обеспечения (ПО). Разработчик путем копирования

* Работа поддержана грантом РФФИ 15-07-07541 А

и дальнейшего изменения некоторого участка кода может получить желаемый результат. Фрагменты кода будем называть *клонами*, если они схожи друг с другом по заданной функции схожести. Клоны могут возникнуть не только в результате копирования неких участков кода. Примером может быть реализация схожей функциональности (многие драйверы в операционной системе делают аналогичную работу). Или ограничения конкретного языка программирования, что не позволяет создать одну общую версию нескольких функций, которые имеют маленькие отличия. Исследования показали, что до 20 процентов исходного кода могут являться клонами [1, 2]. Потребность в поиске клонов кода возникает при поиске функционально похожих частей программы в бинарном или исходном коде программ, при решении задач автоматического рефакторинга, поиска семантических ошибок, возникающих при некорректном копировании участков кода. В данной работе будет описана архитектура инструмента для поиска клонов кода, который обладает высокой точностью и масштабируема. Благодаря ряду новых техник и алгоритмов стало возможно анализировать миллионы строк исходного кода. Описание конкретных алгоритмов будут приведены позже.

2. Типы клонов

Есть три основных типа клонов (классификация приведена из [3]). 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. Где в конкретном фрагменте могут быть также добавлены или удалены некоторые строки.

3. Подходы поиска клонов кода

3.1 Текстовой подход

Алгоритмы поиска клонов кода считают хеш коды одной или нескольких строк исходного кода и сравнивают их. Если хеш коды совпадают, считается, что соответствующие строки исходного кода являются клонами [4]. Алгоритм может объединить последовательные клоны после того, как все клоны найдены. Некоторые алгоритмы могут найти схожие файлы. Для этого рассматривается некое подмножество строк для каждого файла и считают это подмножество его отпечатком. Дальше, если два отпечатка совпадают, считается, что соответствующие файлы похожи [5]. Алгоритмы, работающие на основе этого подхода, находят в основном клоны типа **T1**.

3.2 Лексический подход

Алгоритмы, основанные на этом подходе, в первую очередь получают последовательность лексических единиц (token) путем разбора исходного кода. После чего производится поиск совпадающую подпоследовательность таких единиц. Для этого существуют несколько эффективных алгоритмов, основанных на параметризованном суффиксом дереве [6]. Алгоритмы, работающие на основе этого подхода, в основном находят клоны типа **T1** и **T2**.

3.3 Синтаксический подход

Алгоритмы этого типа работают на абстрактном синтаксическом дереве (Abstract Syntax Tree – AST). Клонами считаются совпадающие AST-поддеревья. Некоторые алгоритмы сразу сравнивают пару деревьев для нахождения совпадающих поддеревьев [7]. Другой алгоритм строит суффиксное дерево (из вершин AST получается последовательность элементов (символы), на основе чего и строится суффиксное дерево) для каждого поддерева AST и сравнивает их [8]. Существует еще один эффективный алгоритм, который строит векторы для каждого AST-поддерева и сравнивает эти векторы [9]. Длина вектора зафиксирована и равна количеству всех возможных типов инструкций в AST. Для конкретного AST поддерева каждый элемент вектора – это количество соответствующих инструкций в этом поддереве. Алгоритмы, работающие на основе AST, находят все три типа клонов.

3.4 Подходы, основанные на метриках

Эти алгоритмы считают ряд метрик для фрагментов кода и сравнивают векторы полученных метрик вместо фрагментов. Обычно метрики считаются на AST или графе зависимостей (PDG) данного фрагмента [10, 11]. Другой метод кластеризует вычисленные метрики с помощью нейронных сетей [12]. Эти методы находят три основные типы клонов. Обычно у этих алгоритмов производительность лучше, чем у алгоритмов, основанных на AST или PDG. Но точность, как правило, гораздо хуже [13].

3.5 Семантический подход

Граф зависимостей программы (PDG) – это объединенный граф потока данных и графа потока управления. Вершины PDG – это инструкции программы, а ребра – зависимости между ними. Есть две основных типов ребер: ребра выражающие зависимости по данным и ребра выражающие зависимости по управлению. PDG наиболее общее представления программы, в нем хранится вся информацию о его семантике и структуре, что позволяет более точно анализировать программу. Оно используется в задачах оптимизации и поиска семантический схожих участков кода (пример PDG-графа приведен на рис.2, пунктирные стрелки это зависимости по

управлению). Алгоритмы, работающие на PDG, пытаются найти изоморфные подграфы в паре PDG-графов [14, 15, 16]. Эти алгоритмы неточные, так как задача изоморфизма графов NP-сложная. Как правило, эти методы обладают большой точностью, но довольно медленны.

4. Сравнения подходов

Описанные выше три типа клонов не могут быть найдены лексическим и текстовым подходом.. Следовательно, эти подходы не могут быть применены к ряду задач таких как, автоматический рефакторинг. Остальные подходы находят все три основные типы клонов. Недостаток подхода основанного на метриках, низкая точность, в результате чего у инструментов использующих этот подход большие погрешности. В общем случае алгоритмы, основанные на PDG, обладают большей точностью, чем методы AST, потому что в AST хранится информация только о структуре программы. PDG содержит информацию о семантике (через потоки данных) и о структуре (через потоки управления) программы, что дает преимущество по сравнению с AST. Следует, что высокая точность инструмента может быт обеспечена только при использования семантического подхода.

5. Модель инструмента поиска клонов кода

Наша модель инструмента поиска клонов кода основана на семантическом подходе, так как цель инструмента – найти все клоны с наибольшей точностью. Инструмент также должен быть масштабируемым для анализа проектов размером в несколько миллионов строк исходного кода. Инструмент состоит из двух основных частей. Первая часть создает PDG-граф из внутреннего представления LLVM (биткода) в ходе компиляции проекта (рис.1.) и реализован как компиляторный проход. Вторая часть инструмента отвечает за анализ PDG-графов в целях нахождения клонов. Она реализована как инструмент пакета LLVM [17].

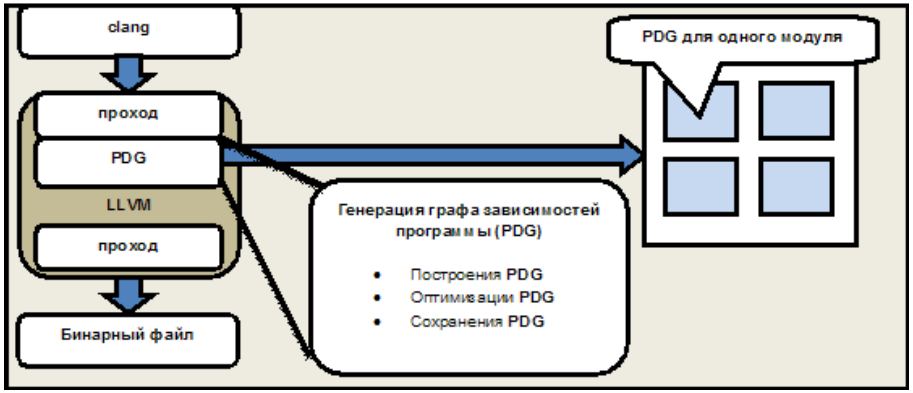

Рис. 1. Архитектура инструмента: генерация PDG.

5.1 Генерация PDG

Во время компиляции генерируется PDG-граф для каждой LLVM-функции [17]. Вершинами PDG являются инструкции LLVM (рис. 2).

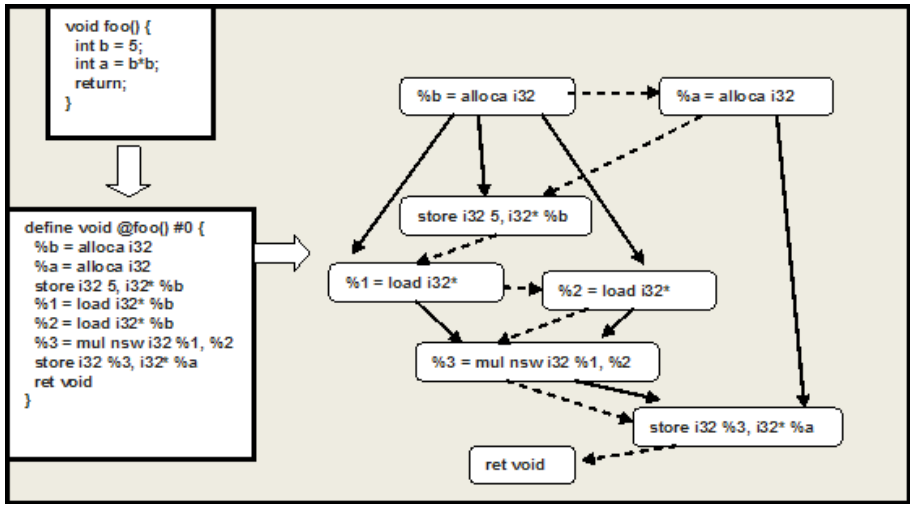

Рис. 2. Пример PDG.

Ребрам соответствуют зависимости по управлению между инструкциями, зависимости, получаемые анализом алиасов и use-def анализом LLVM [17]. Инструмент дает возможность генерировать PDG с тремя разными уровнями детализации. Что дает возможность эффективно искать клоны конкретного типа. В задачах где требуется быстро найти только клоны **T1** и **T2**, используется граф первого уровня. Если требуется дополнительный анализ, например, анализ алиасов, тогда используется граф второго уровня. Для эффективного поиска клонов **T3**, надо использовать полный граф (третьего уровня), что по сравнению с остальными будет работать медленнее. В графе первого уровня включаются только ребра, полученные LLVM use-def анализом. Граф второго уровня также содержит ребра, полученные с использованием анализа алиасов. Максимальную информацию будет содержать граф третьего уровня детализации – в нем есть все ребра первого и второго уровня, а также ребра, отображающие зависимости по управлению. По умолчанию генерируется PDG первого уровня (только на основе LLVM use-def анализа). Для генерации графа второго и третьего уровня детализации предусмотрены отдельные опции. После того, как PDG будет построен, оно оптимизируется и сохраняется в файле. Оптимизации PDG включают в себя:

1. Удаление тех вершин, которые не имеют никаких ребер.

2. Удаление тех вершин, которым не соответствует исходный код. Такие вершины могут возникнуть из-за того, что LLVM bitcode [17] представляется в виде SSA (Single Static Assignment) формы (рис.2 “alloca” инструкция).

5.2 Анализ PDG в целях нахождения клонов кода

Для того, чтобы инструмент был масштабируемым, сохраненные PDG должны быть разделены на части для параллельной обработки. Размер каждой части и количество параллельных процессов зависят от архитектуры машины, на которой будет работать инструмент.

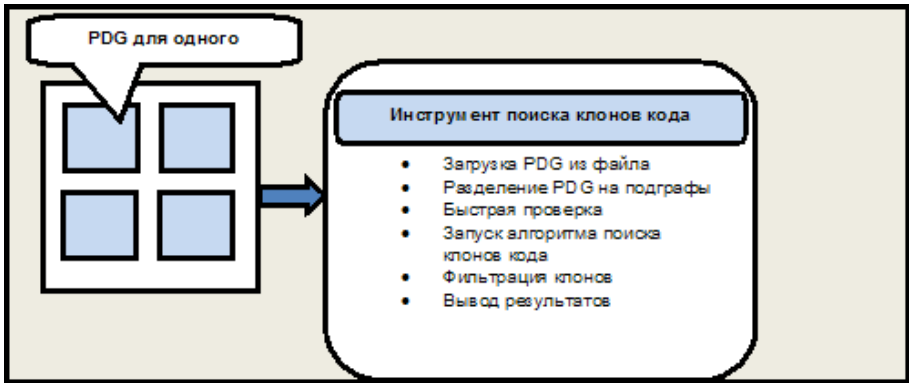

Рис. 3. Архитектура инструмента: анализ PDG.

5.3 Разделения PDG на подграфы

После загрузки PDG в память, они разделяются на единицы сравнения (**EC**) (р.3.). EC-ы представляют собой подграфы PDG и рассматриваются как потенциальные клоны друг друга. Разделение графа на EC-ы должно производится так, чтобы клоны оказались в различных EC-ах. В противном случае в EC попадет только часть клона, в результате чего он найдется частично или вообще не будет найден. Задача состоит в том, что каждый EC полностью содержал потенциальный клон.

5.4 Поиск клонов

После разделения PDG на EC, начинаются их сравнения в целях нахождения клонов. Инструмент содержит два типа алгоритмов сравнения. Первый тип алгоритмов проверяет пару EC на то, что они не клоны (быстрая проверка). Сложность таких алгоритмов O(n)) или O(n*log(n)), где n – количество вершин в обоих EC. Второй тип – это приближенные алгоритмы поиска изоморфных

подграфов. Вычислительная сложность этих алгоритмов довольно велика, она может достигать кубической степени от количества вершин графа. Так как большинство пар ЕС не являются клонами, для них нецелесообразно будет применить алгоритмы второго типа. Таким образом, сначала будут работать алгоритмы первого типа, которые за линейное время, докажут что большинство пар ЕС-ев не клоны. Алгоритмы второго типа будут запущены для тех пар ЕС-ев, которые не были обработаны алгоритмами первого типа. Приблизительно 80 процентов (согласно [1, 2] в среднем клонами может быть до 20 процентов кода) кода будет проанализировано алгоритмами первого типа. Только для малой части будут запущены тяжеловесные алгоритмы второго типа.

5.5 Фильтрация

Последний шаг в процессе поиска клонов кода – фильтрация результатов. Найденные пары изоморфных подграфов дополнительно проверяются алгоритмами фильтрации. Необходимость применения фильтра возникает из-за того, что мы определяем понятие клона для исходного кода программы, а ищем клоны как изоморфные подграфы. Получается, что клон должен быть некой последовательностью строк в файле (не обязательно друг за другом, но обязательно не сильно разбросанным). Цель фильтрации проверить, что исходный код соответствующий изоморфным подграфам не сильно разбросан. Эту проверку необходимо проводить после того как изоморфные подграфы найдены, в противном случае алгоритм поиска клонов может пропустить некоторые клоны кода.

6.Автоматическая генерация тестов

Для проверки точности реализованных алгоритмов разработана система автоматической генерации клонов кода. Для PDG проекта создается новый список графов, в котором попарно объединены графы оригинального проекта (рис.4). После чего алгоритм поиска клонов запускается на паре графов, где первый взят из оригинального списка, а второй из объединенного. Таки образом, «идеальный» алгоритм для всех графов из оригинального списка должен найти клон из объединенного списка. Количество найденных клонов будет характеризовать алгоритм.

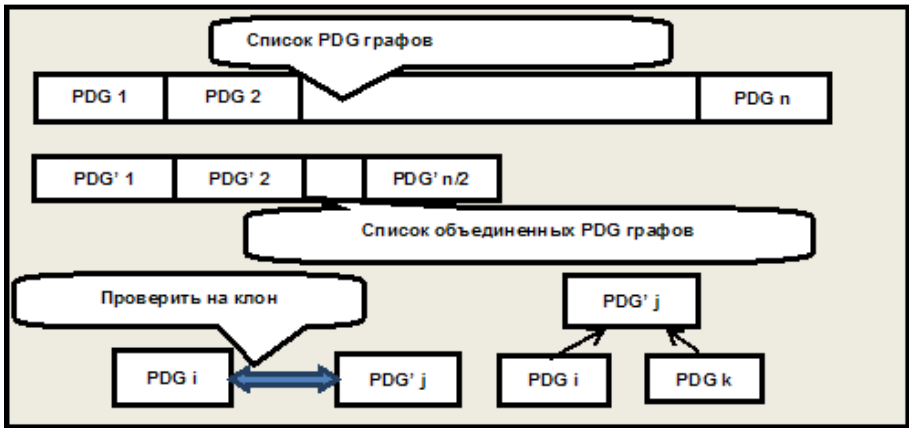

Рис. 4. Схема автоматического тестирования.

7. Заключение

В данной работе рассмотрены существующие подходы к поиску клонов коде. Для достижения высокой точности надо применить семантический подход. Масштабируемость обеспечивается применением двух типа алгоритмов. Первый тип пытается за линейное время определить, что пара PDG не может быть клонами. Второй тип алгоритмов это приближенные алгоритмы поиска изоморфных подграфов. Они запускаются в случае отрицательного результата первого типа алгоритмов. В целях эффективного построения PDG предложен модель инструмента основанной на компиляторной инфраструктуре LLVM, что позволяет получать эти графы в течении компиляции проекта. После чего происходит анализ полученных графов. Так же был предложен метод для автоматического тестирования точности алгоритмов поиска клонов.

Благодаря предложенному подходу, стало возможно сокращение времени генерации PDG. Алгоритмы поиска изоморфных подграфов применяются только для некоторых пар ЕС, что дает возможность создать масштабируемый инструмент поиска клонов кода.

Список литературы

[1]. Baker B., On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95. DOI: 10.1109/WCRE.1995.514697.

[2]. Roy C.K., Cordy J.R., An empirical study of function clones in open source software systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 81-90, DOI: 10.1109/WCRE.2008.54.

[3]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33, 2007, pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[4]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[5]. Manber U., Finding similar files in a large file system, in: Proceedings of the Winter 1994 Usenix Technical Conference, 1994, pp. 2-2.

[6]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[7]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[8]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[9]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[10]. Mayrand J., Leblanc C., Merlo E., Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, 1996, pp. 244-253, DOI: 10.1109/ICSM.1996.565012.

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

[12]. Davey N., Barson P., Field S., Frank R., The development of a software clone detector, International Journal of Applied Software Technology, 1995, Volume 1, no. 3/4, pp. 219-236.

[13]. Gupta S., Gupta P. C., Literature Survey of Clone Detection Techniques, International Journal of Computer Applications, 2014, Volume 99, no. 3, pp. 41-44, DOI: 10.5120/17355-7858.

[14]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[15]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp.301-309, DOI: 10.1109/WCRE.2001.957835.

[16]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[17]. www.llvm.org.

Scalable code clone detection tool based on semantic analysis*

Sevak Sargsyan, <sevaksargsyan@ispras.ru>
Shamil Kurmnagaleev, <kursh@ispras.ru>
Andrey Belevantsev, <abel@ispras.ru>
Hayk Aslanyan, <hayk@ispras.ru>
Artiom Baloian, <artyom@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Annotation. This article describes the methods of code clones detection. New approach of code clones detection is proposed for C/C++ languages based on analysis of existed methods. The method based on semantic analysis of the project, which allows detecting code clones with high accuracy. It is realized as part of LLVM compiler, which allows exceeding existed methods. The tool is consisted of three basic parts. The first part is Program Dependence Graph (PDG) generation and serialization. PDG is constructed during compilation time of the project based on LLVM's intermediate representation. Several simple optimizations are applied on these graphs, then they are serialized to file. The second stage is analyzing of stored PDGs. PDGs are loaded from files and split to subgraphs. Every subgraph is considered as clone candidate. New method is purposed for the splitting, which increases number of detected clones. There are two types of algorithms for clone detection. The first types of algorithms try to prove that the pair of PDGs cannot be clones. These algorithms have linear complexity, which allows processing huge amount of PDGs pairs. In case of failure graph isomorphism algorithms are applied for similar subgraphs detection. The last part is integrated system for automatic testing of algorithm's accuracy. For the project, set of clones are automatically generated, then clone detection algorithms are applied for original source and generated one.

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

DOI: 10.15514/ISPRAS-2015-27(1)-3

For citation: Sargsyan Sevak, Kurmnagaleev Shamil, Belevantsev Andrey, Aslanyan Hayk, Baloian Artiom. Scalable code clone detection tool based on semantic analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, 2015, pp. 39-50 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-3

References.

[1]. Baker B., On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95. DOI: 10.1109/WCRE.1995.514697.

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

[2]. Roy C.K., Cordy J.R., An empirical study of function clones in open source software systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 81-90, DOI: 10.1109/WCRE.2008.54.

[3]. Bellon S., Koschke R., Antoniol G., Krinke J., Merlo E., Comparison and evaluation of clone detection tools, Transactions on Software Engineering 33, 2007, pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[4]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[5]. Manber U., Finding similar files in a large file system, in: Proceedings of the Winter 1994 Usenix Technical Conference, 1994, pp. 2-2.

[6]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[7]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[8]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[9]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[10]. Mayrand J., Leblanc C., Merlo E., Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance, 1996, pp. 244-253, DOI: 10.1109/ICSM.1996.565012.

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

[12]. Davey N., Barson P., Field S., Frank R., The development of a software clone detector, International Journal of Applied Software Technology, 1995, Volume 1, no. 3/4, pp. 219-236.

[13]. Gupta S., Gupta P. C., Literature Survey of Clone Detection Techniques, International Journal of Computer Applications, 2014, Volume 99, no. 3, pp. 41-44, DOI: 10.5120/17355-7858.

[14]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[15]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp.301-309, DOI: 10.1109/WCRE.2001.957835.

[16]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[17]. LLVM www.llvm.org.