УДК 519.172

*Н.Н. Ефанов, А.Р. Федоров, Э.В. Шамбер, А.А. Щербаков, А.П. Елесина*

Московский физико-технический институт (национальный исследовательский университет)

# Pathfinder: статический анализатор программ на базе решения задач достижимости на графах в КС-ограничениях

Рассматривается способ и инструмент детектирования характерных паттернов на графо-структурированных промежуточных представлениях программ посредством решения задачи достижимости с контекстно-свободными(КС) ограничениями. Областями применимости исследования являются статический и гибридный анализ кода, анализ трасс исполнения ПО, графов вызовов. Статья представляет ключевые моменты методики исследования графов программ, проработки архитектурной и функциональной составляющих инструмента анализа. Представлено экспериментальное сравнение базовых алгоритмов проверки КС-достижимости на графах различной связности. Проверка концепции производится на примере детектирования специфической ситуации в работе с массивом данных посредством проведения экспериментов на разработанном программном комплексе. В работе также освещены дальнейшие направления исследования, тестирования и доработки представленного способа и инструмента.

**Ключевые слова:** алгоритмы на графах, контекстно-свободная достижимость, формальные языки, анализ программного обеспечения.

*N. N. Efanov, A. R. Fedorov, E. V. Shamber, A. A. Shcherbakov, A. P. Elesina*

Moscow Institute of Physics and Technology (National Research University)

# Pathfinder: software static analyzer based on solving graph reachability problem with CF-constraints

The method and tool for detecting characteristic patterns on graph-structured intermediate representations of programs by solving the reachability problem with context-free (CF) constraints are considered. The areas of applicability of the study are: static and hybrid analysis of code, software execution traces and call-graphs analysis. Article presents the key points of methodology for studying program graphs, working out the architectural and functional components of the analysis tool. An experimental comparison of basic algorithms for checking CF-reachability (CFL-R) on graphs with different connectivity is presented. Proof-of-concept is carried out by via detecting the specific memory-access patterns during experiments on the developed CFL-R analysis tool. The work also highlights further areas of research, testing and improvement of the presented method and tool.

**Key words:** graph algorithms, context-free reachability, formal languages, software analysis.

## 1. Введение

В настоящее время статический анализ программного обеспечения является устоявшимся и стремительно развивающимся направлением компьютерной науки и системной инженерии, призванным улучшить качество и соответствие программ как функциональным, так и нефункциональным требованиям. Большинство существующих статических анализаторов, разработанных для проверки свойств и поиска ошибок сложных программных систем, представляют собой средства простой проверки программных конструкций, находящие ошибки на основе сопоставления с образцом–паттерном. Такой подход, с одной стороны, выражает баланс между отсуствием строгости и безошибочности анализа, который невозможно произвести абсолютно строго в силу теоремы Райса [1], и вычислительной эффективностью алгоритма анализа. С другой стороны, многие подзадачи статического анализа, такие как «points-to»–анализ и выявление неиспользуемого кода, могут быть решены вычислительно эффективно за счёт проверки достижимости на графо-структурированных представлениях программ, что алгоритмически не сложнее построения транзитивного замыкания входного графа. При этом открываются потенциальные возможности для проведения межпроцедурного анализа и выражения сложных программных паттернов посредством описания путей на графах программ, что теоретически позволяет обобщить методы анализа достижимости с ограничениями на более широкий класс задач. Несмотря на существование множества сложных методов межпроцедурного анализа, не многие из них применяются для практического улучшения инструментов проверки исходного кода, в виду как громоздкости и плохой масштабируемости, так и отсутствия средств и сред, позволяющих облегчить, а в идеале – автоматизировать процесс подготовки программы и описания условий проверки требуемых свойств. Целью данной работы является аггрегация существующих практик анализа графов программ в КС-ограничениях для разработки программного комплекса, который позволит строить на своей базе вычислительно эффективные межпроцедурные статические анализаторы, возможности которых ограничиваются только выразительной способностью КС-языков, описывающих пути в графах программ, генерация коих должна производиться комплексом автоматически.

## 2. КС-достижимость в анализе программного обеспечения

### 2.1. Варианты прикладной задачи анализа КС-достижимости

Введём ряд обозначений. Пусть язык $\boldsymbol{L}(G)$ – язык сконкатенированных меток рёбер ориентированного графа $G = (V, E, L)$, в котором $V, E, L$ – множества вершин, рёбер и меток рёбер соответственно. Пусть $G$ является некоторым исследуемым представлением входной программы $p$: $G = (V, E, L) = G(p)$, $E \subseteq V \times L \times V$, $\boldsymbol{L}(G) = \{w(p)\}$, $w(p) = w(v_0 l_0 v_1, v_1 l_1 v_2, ...), v_i, l_j, v_k \in E$. Рассмотрим следующие задачи:

1) Задача поиска паттернов – найти все помеченные пути графа, содержащие слова некоторого языка $\boldsymbol{L'}(G) \subseteq \boldsymbol{L}(G) : \{P_G^{patterns}\} = \{P_G | w(P_G) \in \boldsymbol{L'}(G)\}$.

2) Задача проверки на анти-паттерн – пусто ли пересечение множества помеченных путей графа с «плохим» языком путей $\boldsymbol{L''}(G)$, недопустимых в программе: $\{P_G \cap \boldsymbol{L''}(G)\}? \equiv \emptyset$.

3) Классическая задача достижимости – существуют ли пути из точки $A$ в точку $B$ в программе?

В терминах задач (1) и (2), условие детектирования некоторой программной конструкции можно описать как требование: задача (1) имеет непустое решение на заданной программе, решение задачи (2) пусто. Паттерны, искомые в задаче (2), назовём ограничивающими, так как их детекция будет означать ограничение выполнения действий над программой – проверка на свойства (1) была выполнена, результат положителен, но свойство (2) запрещает выполнение соответствующего действия, к примеру, применения оптимизации.

### 2.2. Предыдущие работы и основания исследования

Проверка достижимости на графе с ограничениями в терминах формальных языков находит широкое применение в различных задачах компьютерной и программной инженерии, таких как верификация протоколов передачи данных [9], исполнение запросов к графовым БД [8], проверка зависимостей между динамическими библиотеками [5], отдельные направления статического анализа ПО [2, 4, 7, 10–12].

В основе ряда формально-языковых способов и инструментов анализа ПО лежит достижимость с контекстно-свободными ограничениями. В частности, на базе задачи о достижимости в КС-ограничениях исследована и решена проблема связывания символических имён [7], разработан инструментарий для проведения анализа Java-программ на трассах исполнения [2], позволяющий предсказывать диапазон памяти, на которые могут указывать или ссылаться переменные («points-to analysis») [3]. Данный тип анализа может лежать в основе тестирования, отладки, интерпретации и оптимизации ПО. В числе подобных исследований следует также отметить теоретическое исследование по сведению задачи анализа типизированных потоков данных к задаче КС-достижимости [10].

Инструмент Cauli-flower – генератор и решатель задач КС-достижимости общего назначения [4]. Существует несколько примеров его практического использования для построения анализаторов Java-программ и предсказания неиспользуемых регионов памяти посредством анализа промежуточных представлений LLVM [4].К основным недостаткам, затрудняющим практическое применение Cauliflower для анализа ПО, следует отнести абстрагированность от предметной области, отсутствие автоматизации разметки графа и генерации грамматик, необходимость генерации отдельного анализатора для каждой подзадачи.

Инструмент Graspan [11], позиционированный разработчиками как программный комплекс для построения межпроцедурных статических анализаторов кода, использует элементы решения задачи КС-достижимости в исследовании графов программ. Данный инструмент принимает на вход грамматику искомых паттернов в явном виде, что затрудняет построение анализаторов для нетривиальных примеров.

Недостаток необходимости построения грамматик в явном виде устранена в инструменте CoFRA [12], интегрированном в JetBrains ReSharper и протестированном на большой кодовой базе на языке программирования C#, для которого данный инструмент и был разработан. В качестве описания искомых паттернов в нём используются языки МП-автоматов, которые, как известно [15], выразительно эквивалентны КС-языкам.

В силу вышесказанного, требуется предложение способа и разработка инструмента совместного решения задач (1) и (2) пункта 2.1 на промежуточных представлениях программ с учётом существующих практик [4, 11, 12] и перспективных методов анализа КС-достижимости [1]. Анализ промежуточного представления, во-первых, позволяет снизить уровень абстракции по сравнению с исходным кодом, во-вторых, обеспечивает поддержку множества языков программирования, транслируемых в данное представление.

## 3. Pathfinder: набор инструментов для построения анализаторов графов программ в КС-ограничениях

Для решения поставленной задачи авторами разработан набор инструментов «Pathfinder» [13], представляющий собой:

- ядро - CFL-R-решатель — модуль поиска путей в КС-ограничениях
- фронтенд – модули для решения отдельных прикладных задач

Архитектура «Pathfinder» изображена на рис. 1. Входные данные – исследуемая программа и искомые паттерны – подаются на вход соответствующих фронтенд–модулей, где

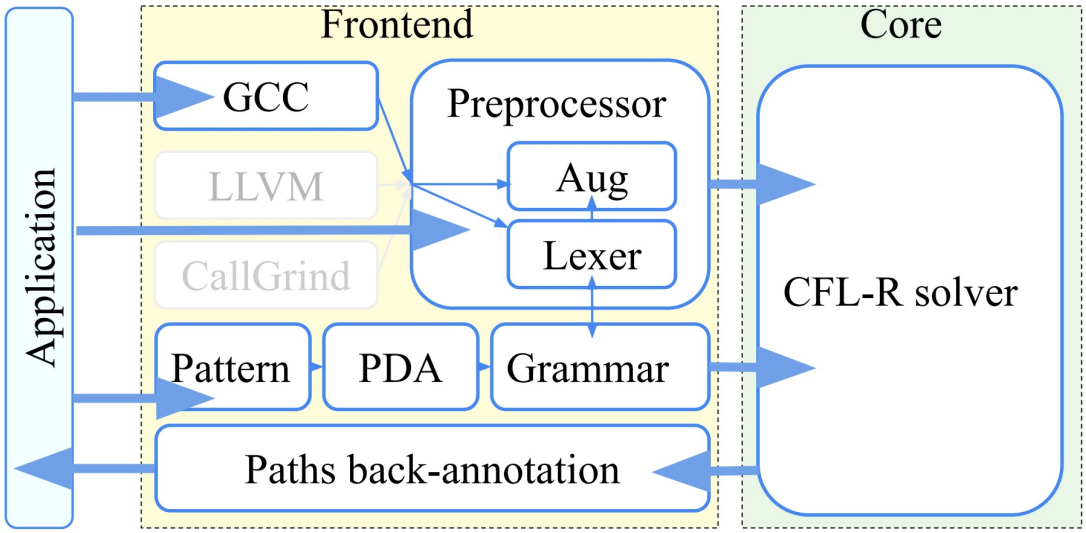



Рис. 1. Архитектура Pathfinder

в зависимости от решаемой прикладной задачи строится графо-структурированное представление программы, его разметка и грамматика паттернов. Далее полученные данные передаются в CFL-R-решатель, осуществляющий поиск путей, соответствующих паттернам. CFL-R-решатель в результате работы возвращает список найденных путей во фронтенд для пост-обработки и анализа результатов.

### 3.1. Подготовка программы к анализу

Подготовка к запуску анализатора на программе заключается в генерации входных данных для соответствующего фронтенд-модуля. К примеру, для использования GCC-модуля в качестве фронтенда, по коду анализируемой программы строится граф потока управления посредством вызова компилятора `gcc` с флагом `-fdump-tree-cfg-graph`. При этом создаются файлы `name.c.012t.cfg.dot`, где каждая функция представлена множеством базовых блоков, соединённых рёбрами передачи управления. Далее полученные `.dot` файлы передаются на вход модуля фронтенда анализатора.

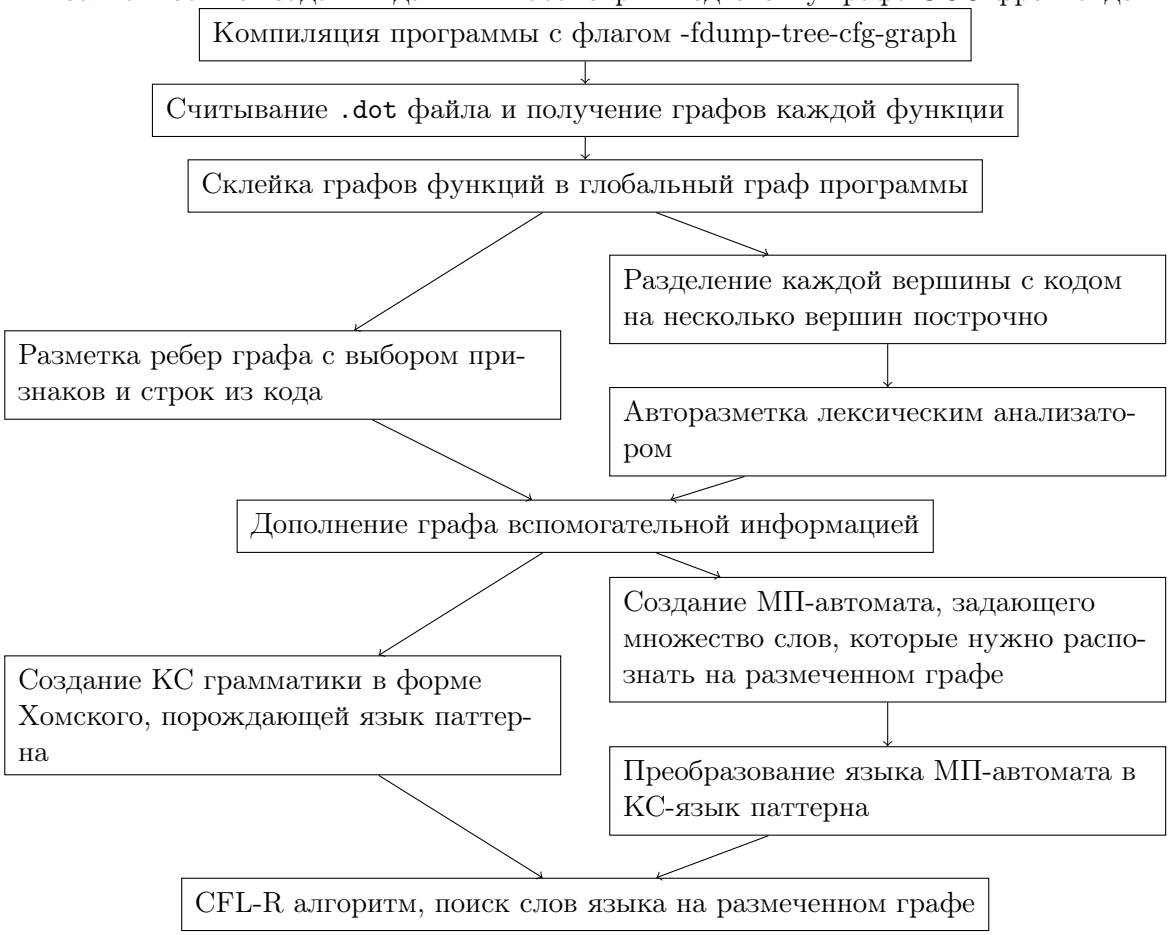
### 3.2. Входные данные

Технически, задачу KC-достижимости можно поставить для любого конечного графа, следовательно, и для любого графо-структурированного промежуточного представления программы на любом языке программирования, транслируемом в данное представление.

В качестве целевого языка программирования используем язык C, а в качестве промежуточного представления – GCC GIMPLE [14], структурированное в виде ориентированного графа, дополненного необходимой информацией (потоки данных для отдельных переменных, метки рёбер, указывающие на семантику переходов между блоками). Модуль анализатора GCC-front считывает `.dot`-файлы, представляя каждую функцию в виде графа, вершинами которого являются базовые блоки. Далее модуль обрабатывает считанное представление программы, сохраняя код базовых блоков и связи между ними для дальнейшей обработки. Искомые паттерны могут передаваться как в формате грамматики, порождающей язык паттернов, так и в виде описания магазинного автомата, распознающего язык паттерна. Способ приведения данных последнего формата к KC-грамматике, реализованный в программном комплексе, описан в 3.5.

### 3.3. Последовательность подготовки к анализу графа программы

Подготовка к анализу происходит в несколько этапов, с возможностью выбора альтернатив в зависимости от задачи и данных. Рассмотрим подготовку графа GCC-фронтендом:




#### Межпроцедурный анализ

Для поддержки межпроцедурного анализа строится общий глобальный граф исполнения программы. Для этого необходимо соединить графы всех функций в один.

Пример для межпроцедурного анализа

```c
#include <stdio.h>

int foo(int a, int b)
{
  if (a == 1)
    return a;
  return a + b;
}

int main()
{
  int a, b;
  a = 1;
  b = 2;
  for (int i = 0; i < 10; i++) {
    printf("%d", foo(a, foo(a, b)));
  }
  return 0;
}
```

Рассмотрим граф, которым GCC представляет вышеуказанный код (рис. 2):

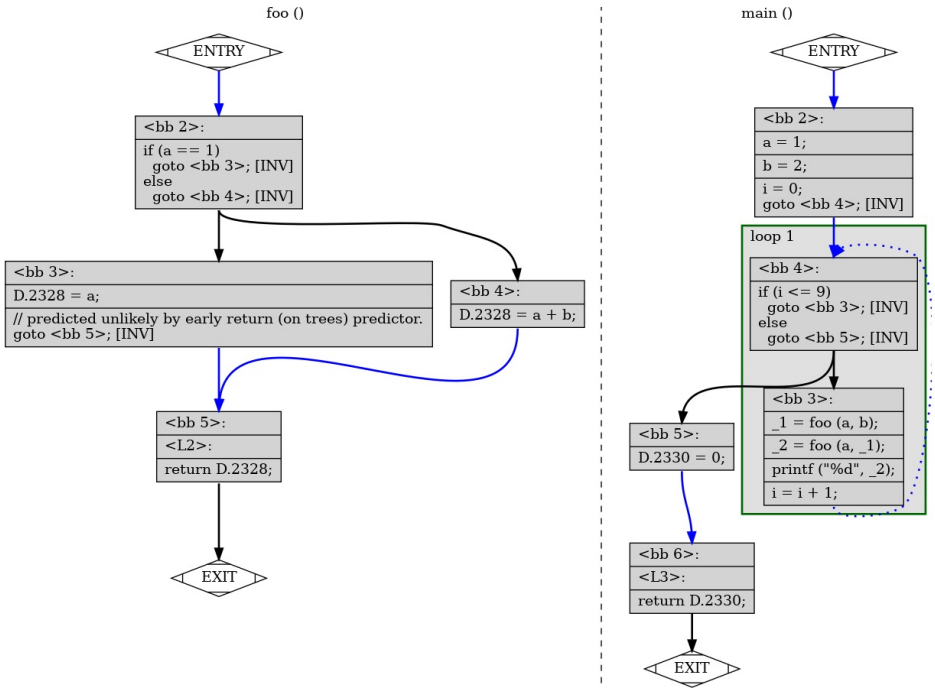



Рис. 2. Представление GCC GIMPLE

Теперь рассмотрим представление этого же графа, получаемое на выходе GCC-модуля (рис. 3)

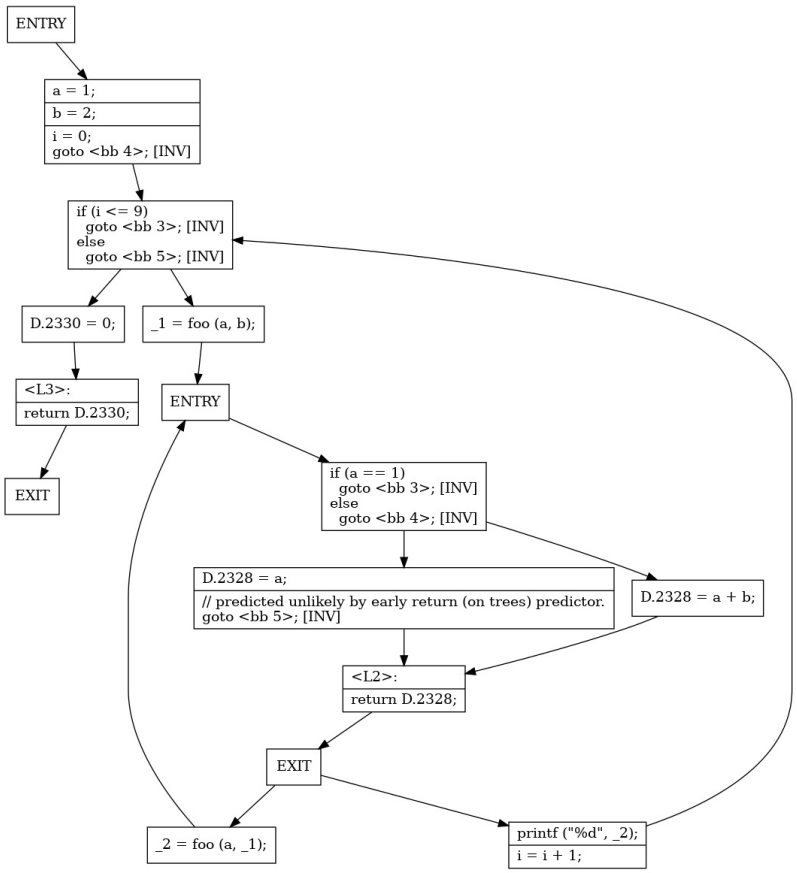



Рис. 3. Стандартный вывод GCC-модуля

Нетрудно заметить, что два независимых графа теперь представлены одним связным, поэтому теперь программу можно анализировать как единое целое. В виду того, что слабая связность и порядок следования базовых блоков по отношению к исходным графам сохраняются, такая трансформация сохраняет чувствительность к потоку.

**Thin-BB: режим «тонких блоков»**

Для упрощения анализа кода парсером базовые блоки разбиваются по строкам. В результате получим новый граф, вершины которого суть строки кода, а ребра сохраняют своё исходное положение, но с соединением строк из одного базового блока, а также с добавлением ребер в функции. Таким образом, сохраняется возможность реализации межпроцедурного анализа посредством отлеживания путей исполнения, проходящих через несколько функций. Приведём пример, как будет выглядеть граф, представленый на рис. 2 и рис. 3, после обработки с опцией `thin-bb` (рис. 4).

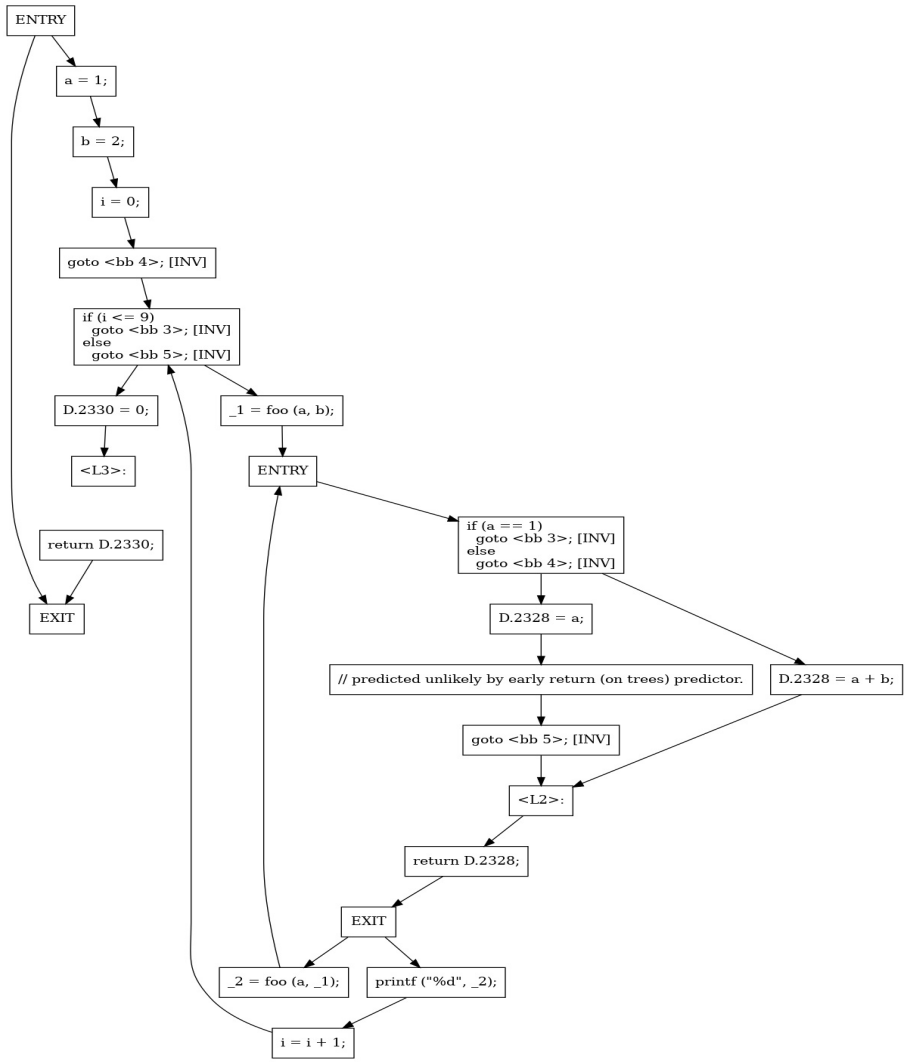



Рис. 3. Thin-BB: режим «тонких блоков»

## 3.4. Лексический анализ меток вершин и разметка рёбер

Лексический анализатор должен формировать из входных последовательностей меток вершин последовательности меток рёбер, которыми будет размечен входной граф. Рассмотрим алфавит меток рёбер $L$, совпадающий с терминальным алфавитом разметки

графа, подаваемого на вход анализатора. Для разметки требуется найти такое отображение $F$ пар меток вершин на метки рёбер, что: $\forall i, j : v_i, v_j \in V \exists a_{ij} \in L \Rightarrow F(v_i.l, v_j.l) = a_{ij}$.

В данной работе отображение $F$ было построено вручную, на основании семантики связей между смежными «тонкими блоками». Для анализа программ Раздела 4 выбраны следующие типы агрументов $F$, соответствующие базовым строкам GIMPLE (табл. 1):

| Тип | Пример | Семантика |
|---|---|---|
| `assign_const` | `D.2679 = 0;` | Присваивание(присв.) константы |
| `assign_var` | `_2 = j;` | Присв. значения переменной |
| `assign_string_const` | `a = "Hello world";` | Присв. указателя на const-строку |
| `assign_function_call` | `buf = malloc (6);` | Вызов функции, присв. |
| `assign_aryphmetic_op` | `_2 = j * j;` | Арифметическая операция, присв. |
| `if_cond` | `if (value != 0)` | Условный оператор |
| `goto` | `goto <bb 4>; [INV]` | Безусловный переход |
| `else` | `else` | Ключевое слово «иначе» |
| `assign_MEM` | `v1 = MEM[(int*)a + 4B];` | Чтение из памяти, присв. |
| `any` | `return D.2109;` | Любой другой тип |

Табл. 1.

Объект, хранящий аргумент, содержит 3 поля: тип, текст метки вершины и словарь свойств `format`, содержащий значимые синтаксические единицы строки GIMPLE, извлечённые из текста метки вершины регулярным выражением. К примеру, для `assign_function_call` ключи словаря – `left, func_name, arguments`. По содержимому словаря возможны проверки предикатов над аргументами $F$, на основании которых производится дополнительная фильтрация вершин, добавление вспомогательных рёбер и атрибутная генерация меток рёбер.

### 3.5. О приведении МП-автомата к КС-грамматике

Для задания грамматики искомых паттернов на практике будет существенно проще строить МП-автомат, нежели выписывать правила вручную, следовательно, разрабатываемый программный комплекс должен иметь функцию приведения МП-автомата к КС-грамматике в нормальной форме Хомского. Опишем способ, которым можно её реализовать.

Заменим каждый переход в автомате $\delta(q, \sigma, z) = (q_1, a_1..a_k)$, где $k > 2$, на несколько переходов вида $\delta(q, \sigma, z) = (q_1, z_1 a_1), \delta(q_1, \varepsilon, z_1) = (q_1, z_2 a_2), .., \delta(q_1, \varepsilon, z_{k-2}) = (q_1, z_{k-1} a_{k-1})$, $\delta(q_1, \varepsilon, z_{k-1}) = (q_1, a_k)$, где $z_1, .., z_{k-1}$ – символы алфавита стека, которые не использовались ранее. Эквивалентность построенного и начального автоматов очевидна. Можно считать, что количество элементов, добавляемое в стек за один переход, не больше константого числа, следовательно, размер нового множества переходов – $O(|\delta|)$. Для построенного автомата воспользуемся теоремой 6.14 из [1], построение займет $O(|\delta|^2)$. Размер грамматики – $O(|\delta|^2)$.

Далее для применения CFL-R алгоритма нужно привести построенную грамматику к нормальной форме Хомского. Для этого воспользуемся алгоритмом, описанным в 7.1 [1].

### 3.6. Решение задачи КС-достижимости: CFL-R алгоритм

Для решения поставленной задачи был использован алгоритм Baseline-CFL-Reachability [15]. К этому алгоритму было добавлено восстановление путей, осуществляемое следующим образом: в ходе выполнения алгоритма сохранялись значения предыдущих вершин после каждого обновления, далее пути восстанавливались рекурсивно.

После дополнения алгоритм Baseline-CFL-Reachability [15] будет выглядеть следующим образом:

```
1  function pathfind(u, S, v):
2      if prev[u][S][v] ≡ (−1, −1, −1) then
3          way ← {u, v};
4          return way;
5      end
6      A_1, A_2, middle ← prev[u][S][v] ;
7      left ← pathfind(u, A_1, middle);
8      right ← pathfind(middle, A_2, v);
9      left.pop();
10     way ← left;
11     for i: right do
12         insert i into way;
13     end
14     return way;
15 W ← H_s ← {(u, A, v) : u -a-> v ∈ S, and A → a ∈ G} ∪ {(u, A, u) : A → ε ∈ G};
16 prev[u][S][v];
17 for all vertexes u, v, each nonterminal S do
18     prev[u][S][v] ← (−1, −1, −1) ;
19 end
20 while W ≠ ∅ do
21     (u, B, v) ← remove from W;
22     for each production A → CB do
23         for each edge (u',C,u) such that (u',A,v) ∉ H_s do
24             insert (u', A, v) into H_s and W;
25             insert (u', A, v, C, B, u) into prev;
26         end
27     end
28     for each production A → BC do
29         for each edge (v, C, v') such that (u, A, v') ∉ H_s do
30             insert (u, A, v') into H_s and W;
31             prev[u][A][v'] ← (B, C, v');
32         end
33     end
34 end
35 for (u, A, v) ∈ H_s such that A = S do
36     way ← {};
37 end
```

**Algorithm 1:** Baseline-CFL-Reachability с восстановлением путей

### 3.7. Анализ используемых алгоритмов

Скорость работы алгоритма зависит от скорости нахождения разности двух множеств, как видно в строках 23 и 29. Предлагается к использованию два разных способа хранения $H_s$. Первый алгоритм использует структуру `std::unordered_set` в качестве контейнера для хранения данных. В таком случае сложность поиска разности множеств равняется $O(n)$, где $n$ - размер множеств. Данный тип хранения позволяет получать выигрыш во времени на разреженных графах (так как размер множеств смежных вершин много меньше числа вершин), каковыми обычно и являются графы потока управления.

Второй алгоритм хранит характеристическую функцию длины порядка $V$, где $V$ - количество вершин в графе. Характеристическая функция разбита на блоки размера $\log_{10}(V)$,

каждый из которых побитово хранится в натуральном числе. За счёт этого сложность поиска разности двух множеств равна $O(\frac{V}{\log_{10}(V)})$. Данный тип хранения даёт преимущество во времени на плотных графах.

Необходимо отметить, что константа в вычислении разности на структурах `std::unordered_set` достаточно велика, в то время как во втором способе хранения константа отсутствует вовсе.

Для наглядности рассмотрим скорость исполнения алгоритмов на разных типах графов (табл. 2, рис. 5, табл. 3, рис. 6, табл. 4, рис. 7, табл. 5, рис. 8).

| Кол-во вершин | Кол-во рёбер | slow | fast |
|---|---|---|---|
| 100 | 200 | 0.001 с. | 0.002 с. |
| 200 | 400 | 0.001 с. | 0.010 с. |
| 300 | 600 | 0.003 с. | 0.013 с. |
| 1000 | 2000 | 0.011 с. | 0.045 с. |
| 10000 | 20000 | 0.05 с. | 3 с. |
| 20000 | 40000 | 0.11 с. | 8.8 с. |
| 50000 | 100000 | 0.31 с. | >60 с. |
| 100000 | 200000 | 0.63 с. | >60 с. |

Табл. 2. Разреженные графы: $E = 2V$

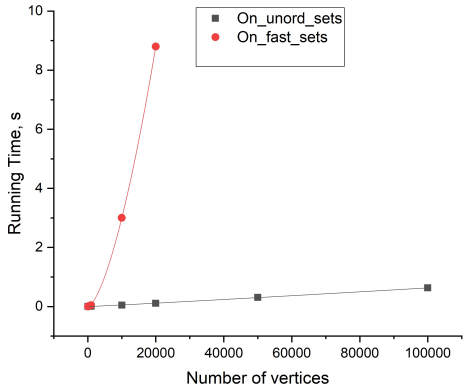



Рис. 5

| Кол-во вершин | Кол-во рёбер | slow | fast |
|---|---|---|---|
| 50 | 350 | 0.035 с. | 0.03 с. |
| 100 | 1000 | 0.17 с. | 0.06 с. |
| 200 | 2830 | 1.27 с. | 0.15 с. |
| 400 | 8000 | 12.5 с. | 0.72 с. |
| 600 | 14700 | 48.2 с. | 1.98 с. |
| 1000 | 31640 | >60 с. | 7.63 с. |
| 1300 | 46870 | >60 с. | 9.91 с. |
| 2000 | 89440 | >60 с. | 31.9 с. |

Табл. 3. Среднесвязные графы: $E \approx V^{1.5}$

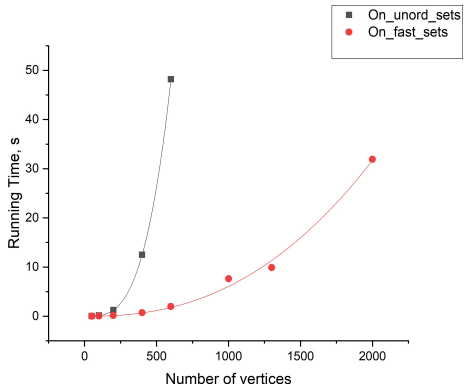



Рис. 6

| Кол-во вершин | Кол-во рёбер | slow | fast |
|---|---|---|---|
| 50 | 625 | 0.055 с. | 0.025 с. |
| 100 | 2500 | 0.28 с. | 0.065 с. |
| 200 | 10000 | 2.28 с. | 0.17 с. |
| 300 | 22500 | 8.63 с. | 0.41 с. |
| 400 | 40000 | 21.5 с. | 0.83 с. |
| 500 | 62500 | 42.7 с. | 1.47 с. |
| 650 | 105625 | >60 с. | 2.85 с. |
| 800 | 160000 | >60 с. | 5.2 с. |
| 1000 | 250000 | >60 с. | 9.46 с. |

Табл. 4. Плотные графы: $E \approx \frac{V^2}{4}$

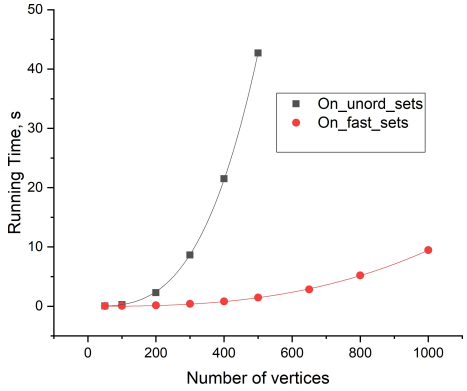



Рис. 7

| Кол-во вершин | Кол-во рёбер | slow | fast |
|---|---|---|---|
| 50 | 2450 | 0.095 с. | 0.023 с. |
| 100 | 9900 | 0.63 с. | 0.072 с. |
| 200 | 39800 | 5.3 с. | 0.22 с. |
| 300 | 89700 | 19.0 с. | 0.60 с. |
| 400 | 159600 | 47.6 с. | 1.28 с. |
| 500 | 249500 | >60 с. | 2.16 с. |
| 650 | 421850 | >60 с. | 4.34 с. |
| 800 | 649200 | >60 с. | 7.86 с. |
| 1000 | 999000 | >60 с. | 14.4 с. |

Табл. 5. Полные графы: $E = V(V-1)$

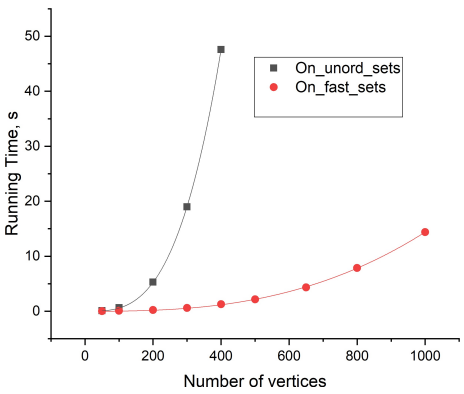



Рис. 8

Эксперименты показывают, что на различных примерах можно добиться улучшения быстродействия анализатора в десять и более раз, используя два алгоритма, вместо одного. Также рассматриваются два разных варианта хранения массива *prev*: `std::unordered_map` и `std::vector`. Первый из них лучше использовать в случае разреженного графа, для того, чтобы не тратить $O(|V|^2)$ на создание массива. В случае же плотных графов лучше использовать `std::vector`, так как таким образом мы сможем избавиться от достаточно большой константы, затрачиваемой на операции `std::unordered_map`.

## 4. Вычислительный эксперимент

Используя разработанный комплекс программ, проведём эксперимент по детектированию характерного паттерна работы с памятью. Рассмотрим два примера: Пример 1 – динамическое выделение памяти для буфера по указателю *buf* с последующей записью в выделенную память, чтением и последующим заполнением буфера нулями.

Пример 1

```
int main()
{
  int* buf=(int*)malloc(4096*sizeof(int));
  buf[2] = 45;
  int k = buf[2] + 1;
  memset(buf, 0, sizeof(buf));
  return 0;
}
```

Пример 2 – динамическое выделение памяти для буфера по указателю *buf*, с чтением и последующим заполнением буфера нулями. Запись в буфер между *malloc* и *memset* отстутствует.

Пример 2

```
int main()
{
  int* buf=(int*)malloc(4096*sizeof(int));
  int v1 = buf[1];
  memset(buf, 0, sizeof(buf));
  buf[2] = 4;
  return 0;
}
```

Положим, существует правило оптимизации $P$ исходной программы: если между выделением и занулением не производилось записи в рассматриваемую память, пару *malloc–memset* можно заменить вызовом *calloc*. Тогда указанное правило может быть применено

к Примеру 2, но не к Примеру 1. Для детектирования подобных ситуаций требуется сформировать паттерны графов программ, дополненных потоками данных для переменных, указывающих в исследуемую область памяти. Паттерны должны:

- Выделять последовательность *malloc–memset*
- Выделять последовательности обращения к исследуемой памяти
- Если производилать запись, паттерн должен быть ограничивающим

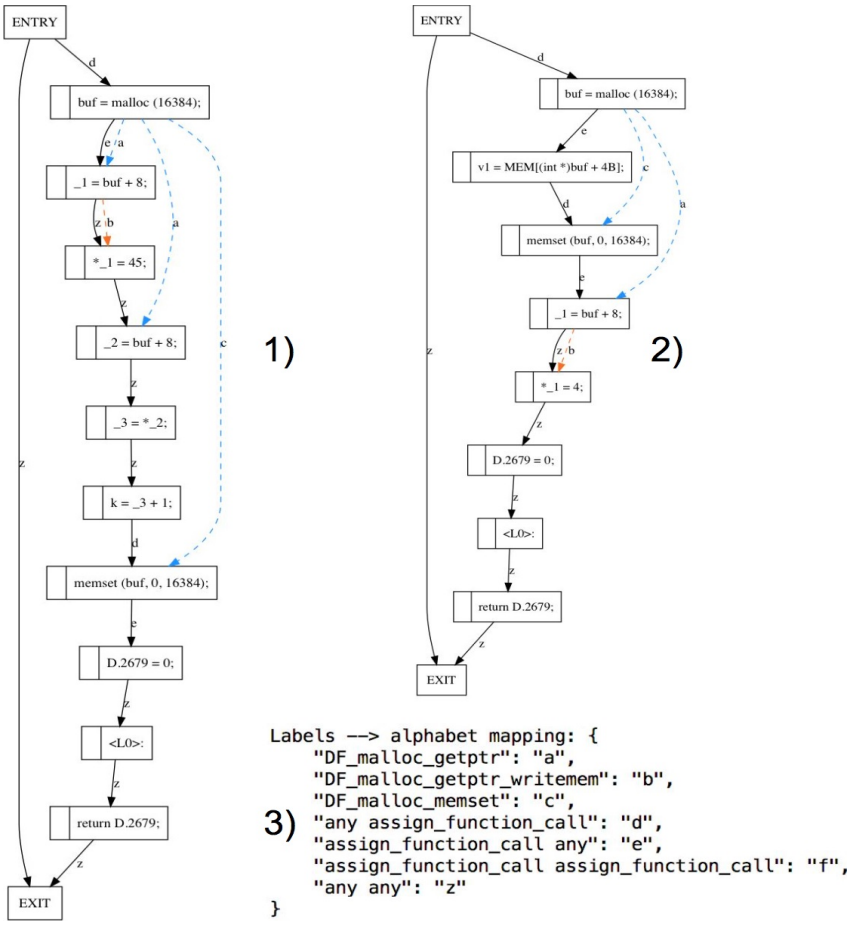



Рис. 9. Графы потока управления, дополненные зависимостями по данным для примеров, содержащих искомый паттерн. Граф (1) содержит ограничивающий паттерн, в отличие от графа (2), следовательно, преобразование $P$ не может быть быть применено к его программе. (3) представляет разметку рёбер.

Построим графы потоков управления исходных примеров в формате «тонких блоков», проведём лексический анализ меток блоков, дополним граф вспомогательными рёбрами и построим метки рёбер в соответствии с отображением пар меток инцидентных вершин (рис. 9).

Сформируем грамматики искомых путей на графе. Для паттерна *malloc–memset* правила грамматики будут иметь вид $\{S \to AB; B \to CB|d; A \to d; C \to e|z\}$ и $\{S \to AB; A \to d; B \to FC; F \to a; C \to HD; H \to b; D \to d|EA; E \to EE|z\}$ по потоку данных и потоку управления соответственно[1]. Правила грамматики ограничивающего паттерна "*malloc*–запись–*memset*"по потоку данных будет иметь вид $\{S \to AB; A \to d; B \to c\}$. Подадим сформированные графы и грамматики на вход анализатора.

[1]Преобразование в нормальную форму Хомского производится программным комплексом автоматически

Для Примера 1 получим вывод (табл. 6):

| Паттерн 1 | Паттерн 2 | Огранич. Паттерн | Описание |
|---|---|---|---|
| 0 8<br>0 2 8<br>1 | 0 8<br>0 2 3 4 5 6 7 8<br>1 | 0 8<br>0 2 3 4 5 6 7 8<br>1 | начало и конец пути<br>последовательность вершин<br>количество путей |

Табл. 6.

Для Примера 2 (табл. 7):

| Паттерн 1 | Паттерн 2 | Огранич. Паттерн | Описание |
|---|---|---|---|
| 0 4<br>0 2 4<br>1 | 0 4<br>0 2 3 4<br>1 | 0 | начало и конец пути<br>последовательность вершин<br>количество путей |

Табл. 7.

Вывод: искомые паттерны содержатся в обоих примерах, начала и концы найденных путей совпадают, а ограничивающий путь содержится только в Примере 1. Следовательно, к Примеру 2 можно применить предложенную трансформацию, а к Примеру 1 – нет. Свойство корректно детектировано.

## 5. Заключение

Выполненная работа имеет характер попытки построения простой, гибкой и вычислительно эффективной платформы для анализа программ посредством решения задач КС-достижимости на компиляторном промежуточном представлении. Разработанный в ходе работы программный комплекс детектирует характерные внутрипроцедурные паттерны работы с памятью в тестовых программах на языке C. Свойства анализатора позволяют обобщить его использование и на ряд задач межпроцедурного анализа. Использование данного инструмента ограничено только процедурами извлечения семантики и разметки входного графа, а также выразительностью КС-языков. Чувствительность к потоку обеспечивается процедурой построения исследуемого графа. Чувствительность к контексту строго не исследована, и существенно зависит от выбранного представления программы. Поиск оптимального представления, обеспечивающего баланс между чувствительностью к контексту и размером входных данных является важнейшим теоретическим продолжением работы, так как позволит более строго определить границы применимости предложенного способа и оценить ключевые статистические характеристики анализатора (уровень ложных срабатываний, устойчивость детектирования и пр). Лексикографический анализ и генерация разметки рёбер также является важным направлением,так как требование по автоматизации генерации $F$ порождает ряд сложных теоретических и практических вопросов: использование хеш-функции как отображения $F$, оценка её свойств и практическая реализация – исследование, планируемое авторами к проведению в ближайшее время.

Основной же задачей на данный момент является тестирование предложенного способа и инструмента на различных примерах, требующих межпроцедурного анализа кода. Работы по такому тестированию проводятся в данный момент. Кроме того, проверка эффективности работы анализатора на большой кодовой базе, ускорения и улучшения масштабируемости CFL-R-решателя является одним из важнейших направлений дальнейшего технического исследования, так как для практической применимости требуется эффективная работа инструмента на больших графах.

## Литература


1. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. — СПб.: Вильямс, 2008.
2. *Lu Y., Shang L., Xie X., Xue J.* An incremental points-to analysis with CFL-Reachability // In Proceedings of the 22nd international conference on Compiler Construction (CC'13). — Berlin: Springer-Verlag, 2013. — P. 61–81.
3. *Steensgaard B.* Points-to analysis in almost linear time // In Symposium on Principles of Programming Languages (POPL). — New York: ACM, 1996. — P. 32-41
4. *Hollingum N., Scholz B.* Cauliflower: a Solver Generator Tool for Context-Free Language Reachability // 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning. — 2017. — P. 171-180.
5. *Tang H., Wang D., Xiong Y., Zhang L., Wang X., Zhang L.* Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization // Programming Languages and Systems, ESOP 2017. — Berlin: Springer, 2017.
6. *Yuan H., Eugster P.* An efficient algorithm for solving the dyck-cfl reachability problem on trees // Programming Languages and Systems, ESOP 2009. — Berlin: Springer, 2009. — P. 175–189.
7. *Zhang Q., Lyu M.R., Yuan H., Su Z.* Fast algorithms for dyck-cflreachability with applications to alias analysis // In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13. — New York: ACM, 2013. — P. 435–446.
8. *Jin R., Hong H., Wang H., Ruan N., Xiang Y.* Computing label-constraint reachability in graph databases // In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). — New York: ACM, 2010. — P. 123–134.
9. *Dolev D., Even S., Karp R.M.* On the security of ping-pong protocols // Information and Control. — 1982. — P. 57-68.
10. *Rehof J., Fähndrich M.* Type-base flow analysis: from polymorphic subtyping to CFL-reachability // In Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). — New York: ACM, 2001. — P. 54–66.
11. *Wang K., Hussain A., Zuo Z., Xu G., Amiri Sani A.* Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code // ASPLOS '17. — ACM, 2017. — P. 389–404.
12. *CoFRA: Inter-procedural Code Static Analyzer / JetBrains-Research Repository, 2019 https://github.com/JetBrains-Research/CoFRA*
13. *Pathfinder: CFL-R Static Software Analysis Tool / MIPT-CS Github Repository, 2021 https://github.com/mipt-cs/pathfinder*
14. *GCC GIMPLE Intermediate Representation / GNU Compiler Collection Manual, 2021 https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html*
15. *Chaudhuri S.* Subcubic algorithms for recursive state machines // In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08). — New York:ACM, 2008. — P. 159–169.


## References


1. *Hopcroft J., Motwani, R., Ullman, J.* Introduction to Automata Theory, Languages, and Computation. — SPb: Wiliams, 2008. (In Russian).

**2.** *Lu Y., Shang L., Xie X., Xue J.* An incremental points-to analysis with CFL-Reachability // In Proceedings of the 22nd international conference on Compiler Construction (CC'13). — Berlin: Springer-Verlag, 2013. — P. 61–81.

**3.** *Steensgaard B.* Points-to analysis in almost linear time // In Symposium on Principles of Programming Languages (POPL). — New York: ACM, 1996. — P. 32-41

**4.** *Hollingum N., Scholz B.* Cauliflower: a Solver Generator Tool for Context-Free Language Reachability // 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning. — 2017. — P. 171-180.

**5.** *Tang H., Wang D., Xiong Y., Zhang L., Wang X., Zhang L.* Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization // Programming Languages and Systems, ESOP 2017. — Berlin: Springer, 2017.

**6.** *Yuan H., Eugster P.* An efficient algorithm for solving the dyck-cfl reachability problem on trees // Programming Languages and Systems, ESOP 2009. — Berlin: Springer, 2009. — P. 175–189.

**7.** *Zhang Q., Lyu M.R., Yuan H., Su Z.* Fast algorithms for dyck-cflreachability with applications to alias analysis // In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13. — New York: ACM, 2013. — P. 435–446.

**8.** *Jin R., Hong H., Wang H., Ruan N., Xiang Y.* Computing label-constraint reachability in graph databases // In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). — New York: ACM, 2010. — P. 123–134.

**9.** *Dolev D., Even S., Karp R.M.* On the security of ping-pong protocols // Information and Control. — 1982. — P. 57-68.

**10.** *Rehof J., Fähndrich M.* Type-base flow analysis: from polymorphic subtyping to CFL-reachability // In Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '01). — New York: ACM, 2001. — P. 54–66.

**11.** *Wang K., Hussain A., Zuo Z., Xu G., Amiri Sani A.* Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code // ASPLOS '17. — ACM, 2017. — P. 389–404.

**12.** *CoFRA: Inter-procedural Code Static Analyzer / JetBrains-Research Repository, 2019 https://github.com/JetBrains-Research/CoFRA*

**13.** *Pathfinder: CFL-R Static Software Analysis Tool / MIPT-CS Github Repository, 2021 https://github.com/mipt-cs/pathfinder*

**14.** *GCC GIMPLE Intermediate Representation / GNU Compiler Collection Manual, 2021 https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html*

**15.** *Chaudhuri S.* Subcubic algorithms for recursive state machines // In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '08). — New York:ACM, 2008. — P. 159–169.

*Поступила в редакцию 07.09.2021.*

# Сведения об авторах статей
## (на момент подачи статьи)

**Статический анализатор программ на базе решения задач достижимости на графах в КС-Ограничениях**

*Ефанов Николай Николаевич (к.ф.-м.н., доцент кафедры инофрматики и вычислительной математики МФТИ)* (nefanov90@gmail.com)

*Федоров Артём Рубенович (студент МФТИ, кафедра анализа данных)* (fedorov.ar@phystech.edu)

*Шамбер Элина Владиславовна (студент МФТИ, кафедра вычислительных технологий и моделирования в геофизике и биоматематике)* (shamber.ev@phystech.edu)

*Щербаков Алексей Андреевич (студент МФТИ, кафедра микропроцессорных технологий в интеллектуальных системах управления)* (shcherbakov.aa@phystech.edu)

*Елесина Анна Павловна (студент МФТИ, кафедра мультимедийных технологий и телекоммуникаций)* (elesina.ap@phystech.edu)

## Ссылки на опубликованные статьи
### (в соответствии с ГОСТ Р 7.0.5-2008))

*Ефанов Н.Н., Федоров А.Р., Шамбер Э.В., Щербаков А.А., Елесина А.П.* Pathfinder: статический анализатор программ на базе решения задач достижимости на графах в КС-ограничениях // Труды МФТИ.– 2021.– Т.11, № 1. № 1. — С. 1–15.

*Efanov N.N., Fedorov A.R., Shamber E.V., Shcherbakov A.A., Elesina A.P.* Pathfinder: software static analyzer based on solving graph reachability problem with CF-constraints // Proceedings of MIPT. — 2021. — V. 11, N 1. — P. 1–15.