DOI: 10.15514/ISPRAS-2025-37(1)-2

Организация статического анализа на абстрактных синтаксических деревьях с помощью конечных автоматов

В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Институт системного программирования им. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, Москва, Ленинские горы, д. 1.

Аннотация. В статье описывается способ организации статического анализа на абстрактных синтаксических деревьях (АСД), повсеместно используемого для поиска ошибок кодирования и других, не требующих глубокого анализа семантики программы. Для большинства известных типов ошибок предложена формализация, основанная на *конечных автоматах над деревьями* (КАД), аналогичных НКА и ДКА для регулярных языков над символьным алфавитом. В отличие от теории КАД, разработанной для алфавитов с ограниченной арностью, что на практике фиксирует количество потомков для каждого типа вершины, рассмотрен случай конечного числа потомков без заранее заданного ограничения. Описаны недетерминированные и детерминированные КАД, показана их эквивалентность, замкнутость регулярных языков над деревьями относительно объединения и пересечения, доказана линейная сложность задачи распознавания дерева КАД. Для тех алгоритмов анализа АСД, что не покрываются регулярными языками над деревьями, рассмотрены классы контекстно-свободных языков.

Ключевые слова: статический анализ; абстрактное синтаксическое дерево; конечные автоматы; регулярные языки.

Для цитирования: Игнатьев В.Н. Организация статического анализа на абстрактных синтаксических деревьях с помощью конечных автоматов. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 41–54. DOI: 10.15514/ISPRAS–2025–37(1)–2.

Static Analysis on Abstract Syntax Trees Based on Finite Automata

V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The paper describes a way of organizing static analysis on abstract syntax trees (AST) widely used for finding coding errors as well as errors that do not require deep understanding of program semantics. For most known types of errors, a formalization is proposed that is based on finite automata over trees (FATs), similar to NFAs and DFAs for regular languages over an alphabet. In contrast to the theory of FATs developed for alphabets with bounded arity, which in practice fixes the number of children for each node type, the case of a finite number of children without a priori bound is considered. Nondeterministic and deterministic FAT are described, their equivalence is shown, the closure of regular languages over trees with respect to union and intersection is shown, and the linear complexity of the FAT tree recognition problem is proven. For the AST analysis algorithms not covered by regular languages over trees, context-free languages are considered.

Keywords: static analysis; abstract syntax tree; finite automata; regular languages.

For citation: Ignatiev V.N. Static analysis on abstract syntax trees based on finite automata. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 41-54 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-2.

1. Введение

Абстрактные синтаксические деревья (АСД) в качестве внутреннего представления для статического анализа исходного кода используются очень широко. Во-первых, они очень удобны для поиска ошибок в коде, связанных не столько с нарушениями семантики программы, сколько с неверным использованием конструкций языка программирования (например, неправильной обработкой исключений в операторе try/catch, ошибочным использованием sizeof, использованием сравнения вместо присваивания и пр.). Во-вторых, на АСД ищутся алгоритмические ошибки (некорректное копирование участка кода) и ошибки, требующие знаний о физическом устройстве исходного кода (потенциально неверное выравнивание операторов). В-третьих, информация об исходном коде из АСД может поставляться следующим уровням анализа.

Алгоритмы поиска ошибок на АСД (детекторы), как правило, используют линейные обходы дерева с целью отыскать определенные виды (шаблоны) поддеревьев; эти обходы могут программироваться как непосредственно в компиляторе или статическом анализаторе средствами, предоставляемыми интерфейсами АСД, так и управляться выражениями на специализированных языках запросов, например на Прологе [1], декларативном языке шаблонов [2, 3] и даже SQL-подобном PQL [4]. Из-за того, что каждой ошибке, как правило, соответствует достаточно узкий и вполне конкретный шаблон, количество типов ошибок составляет сотни, а уровень истинных срабатываний обычно приближается к 100%. Однако формализации создания детекторов на АСД и ее математического обоснования не предложено. В работе [5] предложена классификация детекторов на четыре типа, в которых обходы АСД и таблицы символов организованы таким образом, что гарантируется линейная относительно количества узлов сложность детекторов, но не предложен формальный способ построения таких детекторов.

В данной статье (раздел 2) предложена формализация, основанная на *конечных автоматах над деревьями* (КАД), аналогичных НКА и ДКА для регулярных языков над символьным алфавитом. В отличие от теории КАД, разработанной для алфавитов с ограниченной арностью, что на практике фиксирует количество потомков для каждого типа вершины, рассмотрен случай конечного числа потомков без заранее заданного ограничения. Описаны

недетерминированные и детерминированные КАД, показана их эквивалентность, замкнутость регулярных языков над деревьями относительно объединения и пересечения, доказана линейная сложность задачи распознавания дерева КАД. Для тех АСД-алгоритмов, что не покрываются регулярными языками над деревьями, рассмотрены классы контекстно-свободных языков. В разделе 3 кратко описаны АСД-детекторы статического анализатора SharpChecker [6], предлагающие примеры разработанной формализации.

2. Формализация АСД-детекторов через формальные языки и конечные автоматы над деревьями

Большинство детекторов, работающих на уровне анализа АСД, проверяют достаточно простые свойства исходного кода либо некоторые сочетания комбинаций отдельных свойств и их отрицаний. Другими словами, языки проверяемых свойств составляются из языков отдельных свойств с помощью операций объединения, пересечения и дополнения. Таким образом, естественным обобщением для АСД-детекторов является задание формального языка, словами которого являются АСД, удовлетворяющие проверяемому свойству. В этом случае становится возможным адаптировать теории регулярных и контекстно-свободных языков над деревьями [7] для задания детекторов АСД, что позволит вычислить и обосновать алгоритмическую сложность детектора, исследовать выразительную способность рассматриваемых языков, а также обосновать, почему некоторые детекторы лучше реализовать другим способом.

Автоматы и языки на деревьях были предложены [8] в начале 1980-х для задач верификации схем, логики и лингвистики и остаются актуальны в настоящее время. Формализацию в виде поиска шаблонов на деревьях применяют многие области наук, например, поиск по базам данных [9], анализ XML [10], каждая из которых вносит свою специфику в классическую теорию.

В теории языков над деревьями обычно *алфавит* Σ определяется как пара $(\mathcal{F}, O)$ из конечного множества символов $\mathcal{F}$ (типов вершин) и отображения $O: \mathcal{F} \to \mathrm{N}$, задающего *арность* – фиксированное количество потомков для каждого типа вершины. Такое определение соответствует *ранжированному* алфавиту и не применимо для АСД, так как у некоторых типов вершин допустимо произвольное количество потомков. Поэтому опустим ограничение арности и будем считать, что каждый узел дерева имеет произвольное конечное количество потомков, при этом некоторые вершины могут иметь фиксированную арность. Будем обозначать $\mathcal{F}_n$ – множество символов, имеющих арность n, $\mathcal{F}_*$ – множество символов с переменной конечной арностью, тогда в $\mathcal{F}_0$ будут содержаться листья дерева, которые будем называть *константами*. Будем считать, что любой алфавит содержит хотя бы одну константу. Для АСД $\mathcal{F}_0$ включает идентификаторы и константы в программе, $\mathcal{F}_1$ – унарные операции, $\mathcal{F}_2$ – бинарные и т.д. Обозначим $\mathcal{X}$ – множество *переменных*, принимающих значения из $\mathcal{F}_0$, $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$.

Терминалы $T(\mathcal{F}, \mathcal{X})$ будем задавать рекурсивно следующим образом:

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \mathcal{X})$;
- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$;
- $\forall f \in \mathcal{F} \forall t_1, \dots, t_k \in T(\mathcal{F}, \mathcal{X}): f(t_1 \dots t_k) \in T(\mathcal{F}, \mathcal{X})$;
- ничто другое терминалом $T(\mathcal{F}, \mathcal{X})$ не является.

Тогда *конечное дерево t* определяется как терминал $t \in T(\mathcal{F}, \mathcal{X})$, листьями которого являются константы и переменные, а внутренние вершины являются символами положительной арности. Дерево с корнем a и непосредственными поддеревьями-потомками $t_1, \dots, t_n$ будем записывать как $a(t_1 \dots t_n)$; если $a \in \mathcal{F}_0$, то вместо $a()$ будем писать просто a. Пример дерева $a(b(ef)cd(a))$ приведен на рис. 1.

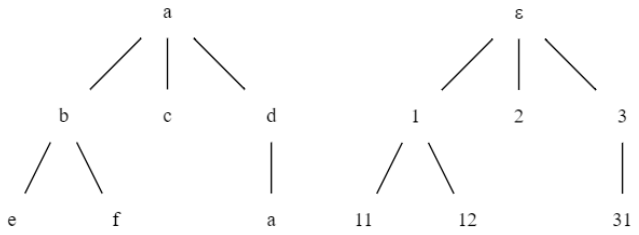

Рис. 1. Дерево $a(b(ef)cd(a))$ и соответствующие позиции вершин.
Fig. 1. The $a(b(ef)cd(a))$ tree and positions of corresponding vertices.

Обозначим N^* – множество всех слов в алфавите натуральных чисел N, построенных из символов алфавита с помощью операции конкатенации $(\cdot)$, ε – пустое слово. Под префиксным порядком $x \le yx, y \in \mathrm{N}^*$ будем понимать $\exists z \in \mathrm{N}^*: y = x \cdot z$. Множество S является *префиксно-замкнутым*, если $x \le y \wedge y \in S \Rightarrow x \in S$. Тогда домен позиций в дереве $\mathcal{P} \subset \mathrm{N}^*$ – конечное непустое префиксно-замкнутое множество, удовлетворяющее условию: $x \cdot y \in \mathcal{P} \Rightarrow \forall i: 1 \le i \le y: x \cdot i \in \mathcal{P}$. В этом случае конечное дерево t может быть задано как отображение $t: \mathcal{P} \to \mathcal{F}$, удовлетворяющее условиям:

- $\forall p \in \mathcal{P}: t(p) \in \mathcal{F}_n, n \ge 1 \Rightarrow \forall i 1 \le i \le n: p \cdot i \in \mathcal{P}$;
- $\forall p \in \mathcal{P}: t(p) \in \mathcal{F}_0 \Rightarrow \forall i \in \mathrm{N}: p \cdot i \notin \mathcal{P}$;
- $\forall p \in \mathcal{P}: t(p) \in \mathcal{F}_* \Rightarrow \exists k \in \mathrm{N}: \forall i, i > k: p \cdot i \notin \mathcal{P}$.

Пример позиций, соответствующих дереву $t = a(b(ef)cd(a))$, приведен справа на рис. 1. Запись $t_i \sim p_i$ будем трактовать как «позиции p_i соответствует вершина t_i».
Упорядоченное множество позиций необходимо для задания порядка обхода дерева. Обычно рассматриваются два порядка:

- снизу-вверх $\uparrow: t(p_1) \prec t(p_2) \Leftrightarrow p_1 > p_2$;
- сверху-вниз $\downarrow: t(p_1) \prec t(p_2) \Leftrightarrow p_1 < p_2$.

Однако при реализации на практике обычно применяется обход дерева в глубину, позволяющий без дополнительных накладных расходов за один обход дерева, обработав каждую внутреннюю вершину дважды, выполнить сразу два порядка обхода: сначала сверху-вниз, а затем снизу-вверх. Будем называть такой обход *двойным* $\downarrow\uparrow$.

2.1 Недерминированные конечные автоматы над деревьями (НКАД)

Определим *недетерминированный конечный автомат на деревьях* как шестерку $A = (Q, \mathcal{F}, Q_f, \delta_\uparrow, \delta_\downarrow, q_0)$, где

- Q – конечное множество состояний автомата;
- $Q_f \subseteq Q$ – множество допускающих состояний;
- $q_0 \in Q$ – начальное состояние;
- $\mathcal{F}$ – алфавит символов вершин дерева (неранжированный);
- $\delta_\uparrow$ – функция, задающая конечное множество переходов снизу-вверх: $\delta_\uparrow: f(R) \to q$, где $f \in \mathcal{F}, q \in Q, R \subseteq Q^*$ – язык, заданный регулярным выражением над множеством состояний;
- $\delta_\downarrow$ – функция, задающая конечное множество переходов сверху-вниз: $\delta_\downarrow: f(q) \to R$.

Автомат начинает работу в состоянии q_0 от корня дерева и двигается: сначала вниз, вычисляя для каждой вершины дерева (позиции) состояние ее потомков в зависимости от ее состояния (Q) и символа ($\mathcal{F}$) в соответствии с $\delta_\downarrow$; а затем вверх, вычисляя состояние родительской вершины по ее символу и состоянию всех потомков, заданному регулярным выражением (R). Очевидно, что возможен случай $\delta_\downarrow = \emptyset$, тогда автомат будет начинать работу от листьев, и ему не требуется начальное состояние q_0, как и наоборот, $\delta_\uparrow = \emptyset$, при котором выполняется только спуск по дереву до листьев.

На рис. 2 представлен алгоритм работы НКАД на дереве. Рекурсивная процедура VISIT реализует обход дерева в глубину, в процессе которого сначала вызываются $\delta_\downarrow(f_t, q)$. Поскольку НКАД недетерминированный, то для пары (f, q) символа вершины и состояния может быть несколько правил, поэтому результирующие r объединяются. Затем рекурсивно обрабатываются поддеревья $c_1, \dots, c_n$. Множество состояний, которым соответствуют c_j, вычисляются как j-й символ всех слов языка $L(r)$. Множества состояний, полученные в результате обхода j-ого поддерева, объединяются в p, а итоговое РВ R строится конкатенацией множеств состояний p потомков f. Далее вычисляется множество состояний вершины f при проходе снизу-вверх как $\delta_\uparrow(f_t, R)$, которое в силу недетерминированности автомата также требует объединения. Полученное множество состояний q вершины f дерева является результатом функции VISIT.

```
function VISIT (T f(c_1 ... c_n), Q q)
    ▷ c_1, ..., c_n – поддеревья, потомки текущего узла с символом f
    ▷ q – текущее состояние НКАД
    {r} ← ∅                          ▷ r ⊆ Q* – РВ, задающее мн-во состояний
                                     c_1, ..., c_n на обходе сверху-вниз
    for all δ ∈ δ↓(f, q) do
        {r} ← {r} ∪ δ(f, q)
    R ← ε                            ▷ r ⊆ Q* – РВ, задающее мн-во состояний
                                     c_1, ..., c_n на обходе снизу-вверх
                                     q ← ∅
    for all r ∈ {r} do
        for j = 1, ..., n do
            p ← ∅                    ▷ p ⊆ Q – мн-во состояний потомка t_j
            for all q_i ∈ Q: q_i ∈ L(r)[j] do    ▷ j-й символ слов языка L(r)
                p ← p ∪ VISIT(c_j, q_i)   ▷ Рекурсивный обход в глубину поддерева
            c_j
            R ← R · p                ▷ Построение R конкатенацией мн-в
                                     состояний всех потомков
        for all δ ∈ δ↑(f, q) do
            q ← q ∪ δ(f, R)          ▷ Мн-во состояний НКАД вершины f
    return q

▷ Запуск алгоритма на дереве t ∈ T
```

if VISIT $(t, q_0) \cap Q_f \neq \emptyset$ **then**
    ACCEPT

Рис. 2. Алгоритм работы НКАД.
Fig. 2. The NFTA processing algorithm.

Теорема 1. Алгоритм на рис. 2 реализует двойной обход, при котором вершины дерева будут обработаны и сверху-вниз, и снизу-вверх.

Доказательство. Утверждение теоремы непосредственно следует из ее псевдокода. Обход сверху-вниз реализуется в первых двух циклах **for all**, гарантирующих, что сначала будет обработана текущая вершина, а потом ее потомки. Это сохраняет частичный порядок на множестве позиций, то есть $p_1 < p_2 \wedge f_1 \sim p_1 \wedge f_2 \sim p_2 \Rightarrow delta_{\downarrow}(f_1, q) \prec delta_{\downarrow}(f_2, q)$. Обход снизу-вверх реализован во втором и третьем циклах. □

Определим, что НКАД A *допускает* дерево t, если НКАД переходит в допускающее состояние из Q_f при обходе любого узла. *Языком* $L(A)$ автомата A будем называть множество всех деревьев, допускаемых A. Множество всех языков, допускаемых НКАД, будем называть *регулярными на деревьях*.

Теорема 2. Допускание в произвольной вершине дерева эквивалентно допусканию в корне дерева.

Доказательство. Пусть НКАД $A = (Q, \mathcal{F}, Q_f, \delta_{\uparrow}, \delta_{\downarrow}, q_0)$. Будем доказывать теорему в обе стороны, то есть:

1. если НКАД A допускает произвольное дерево $t: t \in L(A)$ по определению, то существует НКАД A', в котором корень дерева имеет допускающее состояние после обхода t; и
2. если НКАД A переходит в допускающее состояние в корне произвольного дерева t, то дерево допускается по определению, то есть в произвольной вершине.

Второе утверждение очевидно, поэтому рассмотрим первое. Построим A' следующим образом: $A' = (Q \cup q_e, \mathcal{F}, Q_f \cup q_e, \delta'_{\uparrow}, \delta_{\downarrow}, q_0)$, где $\delta'_{\uparrow}$ включает $\delta_{\uparrow}$, а также следующие правила:

1. $\forall q_f \in Q_f \forall f \in \mathcal{F}: f(Q^* q_f Q^*) \rightarrow q_e$;
2. $\forall f \in \mathcal{F}: f(Q^* q_e Q^*) \rightarrow q_e$.

Таким образом, мы добавили новое состояние $q_e \notin Q$ и правило 2, продвигающее состояние q_e до корня.

Если автомат A допустил в некоторый момент, то либо он допустил в корне, и тогда условие теоремы выполняется, либо с помощью группы правил 1 родительской вершине будет присвоено состояние q_e.

Так как она и последующие вершины на пути к корню не участвовали в выводе, а состояние $q_e \notin Q$, то указанные правила не влияют на работу автомата до допускания дерева t, то есть $L(A) \subseteq L(A')$ и утверждение 1 истинно. □

Рассмотрим способ задания детектора на АСД с помощью НКАД на примере следующей ошибки: «найти в АСД все операции сравнения на равенство и неравенство, операнды которых имеют тип с плавающей запятой».
С помощью НКАД $A = (\{q_0, q_c, q_f, q_E\}, \mathcal{F}, \{q_E\}, \delta_{\uparrow}, \delta_{\downarrow}, q_0)$ зададим язык, выявляющий искомые поддеревья. В таком случае допускание автомата в некоторой вершине дерева будет означать, что она содержит операцию сравнения; допускание позволяет за один проход выявить более одной ошибки.

Предположим, что информация о типах доступна для констант, идентификаторов (из таблицы символов) и возвращаемых значений функций, операция приведения типа имеет следующее представление в дереве: $cast(type\ val)$, где $type$ – идентификатор типа, val – поддерево, задающее приводимое значение. Тогда

$\delta_{\downarrow}$:	$\delta_{\uparrow}$:
$== (q_0 \mid q_c) \to q_c q_c$	$\text{const}_{[type \in \{float, double\}]}(q_c) \to q_f$
$! = (q_0 \mid q_c) \to q_c q_c$	$\text{id}_{[type \in \{float, double\}]}(q_c) \to q_f$
$f(q_c) \to q_c^*, f \in \mathcal{F} \setminus \{==, !=\}$	$\text{call}_{[returntype \in \{float, double\}]}(Q^*) \to q_f$
	$\text{cast}(q_f Q^*) \to q_f$
	$\text{cast}((q_0 \mid q_c) Q^*) \to q_0$
	$== (Q^* q_f Q^*) \to q_E$
	$! = (Q^* q_f Q^*) \to q_E$
	$f(Q^* q_f Q^*) \to q_f, f \in \mathcal{F} \setminus \{\text{cast,call,==,!=}\}$

На пути к листьям потомки операций ==, != помечаются состояниями q_c для повышения производительности, чтобы обрабатывать только те идентификаторы и константы, которые могут участвовать в качестве операндов. На пути снизу-вверх обрабатываются листья, до которых НКАД дошел в состоянии q_c. Если лист, константа или идентификатор переменной или типа имеет тип с плавающей запятой, то НКАД переходит в состояние q_f и распространяет его дальше вверх. Операции вызова, приведения типа (и другие, не приведенные в модельном примере) могут изменить состояние НКАД на начальное. Наконец, если хотя бы один операнд сравнения имеет состояние q_f, автомат допускает, и анализатор может сообщить об ошибке.

2.2 Детерминированные конечные автоматы над деревьями (ДКАД)

Детерминированным конечным автоматом на деревьях будем называть НКАД $A = (Q, \mathcal{F}, Q_f, \delta_{\uparrow}, \delta_{\downarrow}, q_0)$, удовлетворяющий следующим условиям:

1. $f(R_1) \to q_1 \in \delta_{\uparrow} \wedge f(R_2) \to q_2 \in \delta_{\uparrow} \Rightarrow R_1 \cap R_2 = \emptyset \vee q_1 = q_2$;
2. $f(q_1) \to R_1 \in \delta_{\downarrow} \wedge f(q_2) \to R_2 \in \delta_{\downarrow} \Rightarrow R_1 \cap R_2 = \emptyset \vee q_1 = q_2$.

По аналогии с ДКА и НКА на строках и ранжированных алфавитах множество языков, распознаваемых с помощью ДКАД, совпадает с НКАД.

Теорема 3. Любой язык L, распознаваемый НКАД $L = L(N)$, может быть распознан с помощью ДКАД $L(D) \subseteq L(N)$, и обратно.

Доказательство. Пусть НКАД $N = (Q, \mathcal{F}, Q_f, \delta_{\uparrow}, \delta_{\downarrow}, q_0)$. Построим ДКАД $D = (Q', \mathcal{F}, Q_f', \delta_{\uparrow}', \delta_{\downarrow}', q_0)$. Множество состояний ДКАД $Q' = 2^Q$ – множество всех подмножеств Q, состояния которого определим как $q' = \{q_1, \dots, q_n\} \in Q', q_{1 \dots n} \in Q$.

Функцию переходов снизу-вверх $\delta_{\uparrow}'$ определим следующим образом:

$$f(R') \to q' \in \delta_{\uparrow}' \wedge q_1' \dots q_n' \in L(R') \Leftrightarrow$$
$$q' = \{q \in Q \mid \exists q_1 \in q_1', \dots q_n \in q_n', f(R) \to q \in \delta_{\uparrow} \wedge q_1 \dots q_n \in R\}$$

Функцию переходов сверху-вниз $\delta_{\downarrow}'$ определим следующим образом:

$$f(q') \to R' \in \delta_{\downarrow}' \wedge q' = \{q \in Q \mid f(q) \to R \in \delta_{\downarrow}\} \Leftrightarrow$$
$$R' = \bigcup_{q \in q'} f(q)$$

Наконец, $Q_f' = \{q' \in Q' \mid q' = \{q \mid q \in Q \wedge q \in Q_f\}\}$.

Получившийся автомат D' – детерминированный, поскольку:

- для каждого f и каждого q' существует единственное правило перехода в $\delta'_{\uparrow}$, имеющее q' справа, а язык $L(R')$ – регулярный, так как $\$R'$ получен в результате гомоморфизма (подстановки) из R; регулярные языки замкнуты относительно гомоморфизма, поэтому первое условие ДКАД выполняется;
- для каждой пары f и q' будет построено только одно правило в $\delta'_{\downarrow}$, а РВ замкнуты относительно | по определению, поэтому условие 2 ДКАД также выполнено. □

Детерминированный автомат позволяет существенно упростить алгоритм проверки принадлежности дерева языку автомата на рис. 2. Получившийся алгоритм приведен на рис. 3. В детерминированном автомате удается избавиться от цикла по «контекстам» на обходе сверху-вниз, и VISIT возвращает одно состояние, а не множество. НКАД для поиска сравнений нецелочисленных значений на равенство и неравенство, приведенные в качестве примера, удовлетворяют условиям ДКАД.

function VISIT (*T* $f(c_1 \dots c_n), Q\ q$)

▷ $c_1, \dots, c_n$ – поддеревья, потомки текущего узла с символом f

▷ q – текущее состояние ДКАД

$r \leftarrow \delta(f, q)$ ▷ $r \subseteq Q^*$ – РВ, задающее мн-во состояний $c_1, \dots, c_n$ на обходе сверху-вниз

$R \leftarrow \varepsilon$ ▷ $r \subseteq Q^*$ – РВ, задающее мн-во состояний $c_1, \dots, c_n$ на обходе снизу-вверх

for $j = 1, \dots, n$ **do**

$p \leftarrow \emptyset$ ▷ $p \subseteq Q$ – мн-во состояний потомка t_j

for all $q_i \in Q : q_i \in L(r)[j]$ **do** ▷ j-й символ слов языка $L(r)$

$p \leftarrow p \cup \text{VISIT}(c_j, q_i)$ ▷ Рекурсивный обход в глубину поддерева c_j

$R \leftarrow R \cdot p$ ▷ Построение R конкатенацией мн-в состояний всех потомков

$q \leftarrow \delta(f, R)$ ▷ Состояние вершины f

return q

▷ Запуск алгоритма на дереве $t \in T$

if VISIT $(t, q_0) \in Q_f$ **then**

ACCEPT

Рис. 3. Алгоритм работы ДКАД.
Fig. 3. The DFTA processing algorithm.

2.3 Регулярные языки на деревьях

Для задания детекторов на АСД бывает необходимо задавать множество искомых поддеревьев в виде объединения, пересечения и дополнения языков НКАД, выделяющих деревья с определенными свойствами. Покажем, что задаваемые ДКАД языки замкнуты относительно этих операций.

Теорема 4. Класс регулярных языков на деревьях замкнут относительно объединения.

Доказательство. Рассмотрим два произвольных НКАД N_1 и N_2 и покажем, что $L(N_1) \cup L(N_2)$ – регулярный язык на дереве. По теореме 3 существуют ДКАД $A_1 = (Q^1, \mathcal{F}^1, Q_f^1, \delta_\uparrow^1, \delta_\downarrow^1, q_0^1) : L(A_1) = L(N_1)$ и ДКАД $A_2 = (Q^2, \mathcal{F}^2, Q_f^2, \delta_\uparrow^2, \delta_\downarrow^2, q_0^2) : L(A_2) = L(N_2)$. Построим НКАД $N = (Q, \mathcal{F}, Q_f, \delta_\uparrow, \delta_\downarrow, q_0) : L(N) = L(A_1) \cup L(A_2)$, реализующий объединение языков автоматов A_1 и A_2.

Поскольку НКАД должен допускать деревья обоих языков, то алфавит $\mathcal{F} = \mathcal{F}^1 \cup \mathcal{F}^2$. Без ограничения общности можно считать, что $Q_1 \cap Q_2 = q_0$, иначе можно переобозначить состояния таким образом, чтобы они все различались, кроме q_0. Тогда $Q = Q_1 \cup Q_2, Q_f = Q_f^1 \cup Q_f^2, q_0 = q_0^1 = q_0^2$. Функции переходов определим аналогично как объединение $\delta_\uparrow = \delta_\uparrow^1 \cup \delta_\uparrow^2, \delta_\downarrow = \delta_\downarrow^1 \cup \delta_\downarrow^2$.

Язык автомата $L(A) = L(A_1) \cup L(A_2)$, что можно легко показать тем, что $\forall t \in L(A) : t \in L(A_1) \cup L(A_2)$ и обратно. Если $t \in L(A)$, то существует цепочка конфигураций A, обеспечивающая допускание t автоматом A. Поскольку в A не содержится переходов между состояниями автоматов A_1 и A_2, то эта последовательность конфигураций воспроизводима без изменений в A_1 или A_2, то есть слово t принадлежит языку $L(A_1)$ или $L(A_2)$. И обратно, вывод произвольного дерева, допускающегося в A_1 или A_2, очевидно, допустим в A.

Можно заметить, что построенный НКАД A будет недетерминированным, если, например, для $f(q_0) \to R_1 \in \delta_\downarrow^1 \wedge f(q_0) \to R_2 \in \delta_\downarrow^2 \wedge R_1 \cap R_2 \neq \emptyset$. □

Будем называть НКАД A *полным*, если $\forall f \in \mathcal{F} \exists f(R) \to q \in \delta_\uparrow \wedge \exists f(q) \to R \in \delta_\downarrow$. Любой НКАД можно дополнить «мертвым» состоянием, в котором он целиком обойдет любое дерево, поданное ему на вход, но не допустит его, определив все недостающие переходы переводящими автомат в «мертвое» состояние и никогда его не покидающим.

Теорема 5. Класс регулярных языков на деревьях замкнут относительно дополнения.

Доказательство. Для произвольного регулярного языка L существует дополненный ДКАД $A = (Q, \mathcal{F}, Q_f, \delta_\uparrow, \delta_\downarrow, q_0) : L(A) = L$. Построим автомат $A' = (Q, \mathcal{F}, Q_f', \delta_\uparrow, \delta_\downarrow, q_0)$ для языка $\bar{L} = \mathcal{F}^* \setminus L$, который будет иметь то же множество состояний, алфавит, начальное состояний и те же функции переходов сверху-вниз и снизу-вверх, что и A. А множество допускающих состояний $Q_f' = Q \setminus Q_f$ будет состоять из всех недопускающих состояний в A. Такой автомат будет допускать любое дерево, которое не допускал ДКАД A, то есть язык $\bar{L}$. При этом «мертвое» состояние A, при наличии, станет допускающим состоянием в A', поэтому необходимо дополнить НКАД A первым шагом. □

Теорема 6. Класс регулярных языков на деревьях замкнут относительно пересечения.

Доказательство. Можно непосредственно выразить пересечение через дополнение и объединение: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, после чего свести теорему к комбинации уже доказанных теорем 4 и 5. Однако возможно также построение НКАД $A = (Q, \mathcal{F}, Q_f, \delta_\uparrow, \delta_\downarrow, q_0)$ для языка $L(A) = L(A_1) \cap L(A_2)$ для автоматов, заданных при доказательстве теоремы 4. Состояниями автомата будут все возможные пары, первым элементом которых является состояние A_1, а вторым – состояние A_2, а множество допускающих состояний будет состоять из всех пар, соответствующие элементы которых входят в Q_f^1 и Q_f^2.

$$Q = \{q | q = q_1 \times q_2, q_1 \in Q^1, q_2 \in Q^2\}$$
$$\mathcal{F} = \mathcal{F}^1 \cup \mathcal{F}^2$$
$$Q_f = \{q | q = q_1 \times q_2, q_1 \in Q_f^1, q_2 \in Q_f^2\}$$
$$q_0 = q_0^1 \times q_0^2$$
$$\delta_\uparrow = \delta_\uparrow^1 \times \delta_\uparrow^2 = \{f(R_1 \times R_2) \to q_1 \times q_2 \big| f(R_1) \to q_1 \in \delta_\uparrow^1, f(R_2) \to q_2 \in \delta_\uparrow^2\}$$

$$\delta_{\downarrow} = \delta_{\downarrow}^{1} \times \delta_{\downarrow}^{2} \ \square$$

Сформулированные теоремы 4-6 означают возможность конструировать детекторы с использованием операций объединения, дополнения и пересечения регулярных языков на деревьях, сохраняя сложность детекторов для отдельных языков.

Теорема 7. Сложность распознавания регулярных языков на дереве линейна относительно количества вершин дерева.

Доказательство. Теорема означает, что для любого языка на дереве, для которого можно построить НКАД, существует алгоритм, реализующий его проверку за линейное относительно количества вершин дерева время. По теореме 3 существует ДКАД, допускающий тот же язык. Алгоритм рис. 3, реализующий проверку принадлежности дерева языку, заданному ДКАД, имеет линейную сложность, в чем легко убедиться непосредственно по реализации. Таким образом, задача распознавания регулярного языка на дереве имеет линейную сложность. □

К сожалению, далеко не все ошибки на АСД можно задать с помощью регулярного языка на деревьях, даже если для их проверки не требуется построения никаких других представлений программы. Одним из примеров такой ошибки является поиск похожих участков кода, который требуется для эвристического алгоритма поиска упомянутых во введении ошибок некорректного копирования участка кода. В типичном случае такой ошибки некоторый фрагмент кода копируется, и в копии все использования некоторой переменной, кроме одного, заменяются на другую. Другим примером является поиск совпадающих ветвей условного оператора, switch-блока, идентичных тел функций. Для реализации всех перечисленных детекторов необходима «память», которая позволит поэлементно сравнить два поддерева, что не может быть реализовано с помощью НКАД.

Теорема 8. Не существует регулярного языка для поиска совпадающих поддеревьев.

Доказательство. Будем доказывать методом от противного. Пусть существует НКАД A, допускающий произвольное дерево t тогда и только тогда, когда в нем есть идентичные поддеревья. Пусть A имеет n состояний. Рассмотрим дерево $f_1(f_2(t_1 \dots t_m)f_2(t_1 \dots t_m))$, представленное на рис. 4, в котором $m > n$ и $\forall i,j \in [1,m]: i \neq j \Rightarrow t_i \neq t_j$.

Для того, чтобы убедиться, что поддеревья вершины f_1 совпадают, на проходе снизу-вверх необходимо закодировать поддерево $f_2(t_1 \dots t_m)$. Единственным способом это сделать является переход в некоторое состояние в зависимости от состояний поддеревьев. Поскольку все t_i различны, они должны представляться различными состояниями $q_1 \dots q_m$, а в вершине f_2 автомат должен быть в новом состоянии q_{m+1}, однако число состояний НКАД по предположению $n < m$, поэтому существует как минимум пара состояний $q_i, q_j \in q_1 \dots q_{m+1}$, которые совпадают, а, следовательно, существуют различные поддеревья, которые НКАД не может различить, что противоречит предположению. □

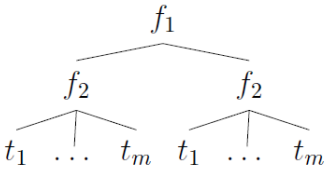

Рис. 4. Дерево $f_1(f_2(t_1 \dots t_m)f_2(t_1 \dots t_m))$.
Fig. 4. The $f_1(f_2(t_1 \dots t_m)f_2(t_1 \dots t_m))$ tree.

Теорема 8 демонстрирует пример класса нерегулярных языков на деревьях, для которых не существует реализации линейной относительно размера входного дерева, поэтому на

практике они реализуются либо с помощью эвристических алгоритмов на АСД, либо с помощью других подходов, например, на основе машинного обучения.

3. Реализация статического анализа АСД в инструменте SharpChecker

В анализаторе SharpChecker на уровне анализа АСД реализовано более 70 детекторов. Отношение количества ошибок к количеству детекторов – «многие ко многим», поскольку один тип ошибки может быть обнаружен несколькими способами, и наоборот, один алгоритм может находить несколько различных типов ошибок. Большая часть обнаруживаемых ошибок может быть реализована с помощью ДКАД, однако на практике это не всегда целесообразно. Существование реализации с помощью регулярного языка означает наличие эффективной реализации за линейное время относительно размера АСД, однако в некоторых случаях бывает удобно, например, сразу или неоднократно обойти поддерево ограниченного размера на проходе сверху-вниз, вместо реализации двух обработчиков для $\delta_{\downarrow}$ и $\delta_{\uparrow}$.

Упомянутые детекторы были запущены на наборе открытых проектов из 6 миллионов строк кода на языке C#. Всего встретились ошибки 47 типов. Детекторы ошибок, для которых существуют регулярные языки, имеют близкую к 100% долю истинных предупреждений. Это объясняется простотой их реализации. Ложные предупреждения, как правило, объясняются ошибкой в реализации детектора, из-за которой не учитываются некоторые шаблоны кода, и могут быть исправлены. Критерии истинности таких детекторов обычно несложно формализуются. Однако на практике для сокращения количества нежелательных предупреждений применяются эвристики для выявления наиболее серьезных проблем и подавления несущественных.

На практике ошибки, которые могут задаваться регулярными языками на АСД, хорошо подходят для безопасного расширения анализатора пользовательскими детекторами, поскольку никакая ошибка в реализации не приведет к зависанию или замедлению анализа на произвольном АСД. Для задания регулярных языков в символьном алфавите применяются регулярные выражения и грамматики, а для ДКАД на дереве необходимо разработать аналогичную нотацию, которая позволит, по аналогии с анализом помеченных данных, добавлять и редактировать правила детекторов без пересборки анализатора. Другим важным преимуществом такого подхода является возможность использовать одну запись детектора для нескольких похожих языков путем реализации движка над унифицированным АСД.

Наиболее сложную реализацию имеют те АСД-детекторы, для формализации которых не существует регулярного языка на дереве. Как правило, для их реализации требуется сбор информации по всей программе либо многократный обход поддеревьев АСД. Реализация использует в качестве точек входа тот же обход АСД в глубину, однако вместо поиска определенных сигнатур деревьев они накапливают информацию для последующего анализа. После окончания фазы анализа АСД для C#-проекта или всей сборки ошибки выдаются в результате анализа собранной информации.

В анализаторе SharpChecker двадцать таких детекторов. Они были запущены на том же наборе проектов. Точность детекторов в среднем оказывается ниже, чем у детекторов регулярных языков, но все еще выше, чем у детекторов, реализованных не на АСД. Причины ложных срабатываний чаще всего объясняются несовершенством используемых эвристик, необходимых для обеспечения масштабируемости или повышения точности. В качестве примеров используемых эвристик рассмотрим алгоритмы работы отдельных детекторов.

Алгоритм поиска ошибок некорректного копирования (`BAD_COPY_PASTE`), упомянутых выше, следующий. Сначала выполняется построение цепочек токенов для близких в коде `if`-блоков и поэлементное их сравнение. Если две цепочки совпадают на заданный процент токенов, то в них выполняется поиск подстановок для всех идентификаторов. Если некоторая переменная v во всех случаях, кроме одного, заменена на другую c, то предполагается, что

может быть ошибка при использовании v. Проверяется совместимость типов заменяемой переменной v и кандидата c, выполняется поиск мест использования замены c, чтобы убедиться, что переменная еще встречается в анализируемом `if`-блоке. Даже такой несложный алгоритм позволяет находить интересные алгоритмические ошибки в работающем коде популярных проектов, которые не были обнаружены при тестировании. Однако использование эвристик существенно ограничивает полноту анализа и критически необходимо для обеспечения масштабируемости, потому что, например, поэлементное сравнение всех `if`-блоков в программе неприемлемо на практике. Для задач, в которых необходим поиск похожих фрагментов кода (клонов), активно используются алгоритмы на основе машинного обучения [11, 12].

Ошибка `FORGOTTEN_READONLY` сообщает о необходимости добавить ключевое слово `readonly` в объявление данной поля класса, поскольку его значение нигде не меняется в анализируемом коде и не может быть изменено из внешнего кода из-за ограничений доступа (например, `private`-поле `private`-класса). Данная ошибка является скорее дефектом кода и практически не влияет на выполнение программы, однако реализована как побочный эффект выполнения анализа для поиска полей класса, имеющих константное значение (для повышения точности работы символьного выполнения в инструменте SharpChecker, выходящего за рамки данной статьи). Для реализации такого анализа необходим просмотр всех конструкций, которые могут изменить значение переменной в языке – присваиваний, инкрементов-декрементов, передачу в качестве аргументов с модификаторами `ref`, `out` и т.п. – и сохранение информации для всех использований поля класса в том числе в разных компиляциях (для разных исполняемых файлов и библиотек).

Ошибка `OFF_BY_ONE` демонстрирует, как с помощью эвристик, не интерпретируя исходный код, можно искать алгоритмические ошибки, основываясь на шаблонах частых ошибок. Данный детектор обнаруживает циклы, обрабатывающие массивы и другие контейнеры, в которых выполняются обращения ко всем, кроме крайних элементов, что часто является следствием ошибки в инициализаторе или условии выхода из цикла. Другим шаблоном ошибки является, наоборот, выход за границы контейнера из-за некорректных условий цикла. Примеры таких циклов приведены далее:

- `for (int i = 0; i <= x.Length; i++)`
- `i = 0; while (i <= x.Length)`
- `for (int i = 0; i < x.Length - 1; i++)`
- `for (int i = 1; i < x.Length; i++)`, `for (int i = 1; i <= x.Length - 1; i++)`
- `for (int i = x.Length - 1; i > 0; i--)`

Для поиска таких ошибок недостаточно анализа только заголовка цикла, потому что распространенным случаем является обработка крайних элементов за пределами цикла; такой шаблон для большинства случаев поддержан в анализаторе. Детектор выдает ~30% ложных предупреждений для случаев, когда алгоритм действительно требует отдельного анализа крайних элементов, который был не распознан. Это означает, что детектор можно улучшить, однако доля ложных срабатываний будет сокращаться непропорционально затраченным усилиям, а 100%-точности достичь не удастся, так как существуют алгоритмы, нарушающие гипотезу детектора – то есть намеренно не обрабатывающие все элементы контейнера.

4. Заключение

В работе представлена формализация для АСД-детекторов статического анализа, основанная на конечных автоматах над деревьями. Рассмотрен случай конечного числа потомков без

заранее заданного ограничения. Описаны недетерминированные и детерминированные КАД, показана их эквивалентность, доказана линейная сложность задачи распознавания дерева КАД. Для тех алгоритмов анализа АСД, что не покрываются регулярными языками над деревьями, рассмотрены классы контекстно-свободных языков.

В инструменте анализа SharpChecker алгоритмы, которые реализованы с помощью ДКАД, демонстрируют почти стопроцентную долю истинных срабатываний. В детекторах, не покрывающихся регулярными языками, истинных срабатываний меньше, но по-прежнему больше, чем в детекторах на основе символьного выполнения.

Разнообразие эвристик в детекторах, не покрывающихся ДКАД, не позволяет их реализовать с помощью НКАД, так как они часто гораздо сложнее шаблона искомой ошибки, а также выведены разработчиками детектора на основе ручного анализа и систематизации результатов детектора на миллионах строк кода. Анализ АСД позволяет извлечь лишь малое количество информации о программе, которого недостаточно для доказательства наличия или отсутствия ошибки искомого типа, поэтому для практического применения созданного детектора необходимо достижение высокой точности (более 70%). Сейчас это достигается за счет количества исследованного кода – как и в методах машинного обучения, эвристики оптимизируются на входных данных. Другим способом является проверка найденных предупреждений с помощью машинного обучения [13] или последующими этапами анализа [14], обладающими более полной информацией о программе. Это позволяет сократить ресурсоемкость за счет предварительного легковесного отбора участков кода, в которых ошибка наиболее вероятна.

Список литературы / References

[1]. Marpons, G., Mariño, J., Carro, M., Herranz, Á., Moreno-Navarro, J.J., Fredlund, LÅ. (2007). Automatic Coding Rule Conformance Checking Using Logic Programming. In: Hudak, P., Warren, D.S. (eds) Practical Aspects of Declarative Languages. PADL 2008. Lecture Notes in Computer Science, vol 4902. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-77442-6_3.

[2]. С.В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST. Труды Института системного программирования РАН, т. 20, 2011, с. 51-68.

[3]. Tomoko Matsumura, Akito Monden, and Ken-ichi Matsumoto. 2002. A method for detecting faulty code violating implicit coding rules. In Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02). Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/512035.512040.

[4]. Stan Jarzabek. – «Design of flexible static program analyzers with PQL». – B: IEEE Transactions on software engineering 24.3 (1998), pp. 197– 215.

[5]. А. А. Белеванцев. Многоуровневый статический анализ исходного кода программ для обеспечения качества программ. Программирование, 2017, т. 43, № 6, с. 3-26.

[6]. V.K. Koshelev, V.N. Ignatiev, A.I. Borzilov и A.A. Belevantsev. – «SharpChecker: Static analysis tool for C# programs». – B: Programming and Computer Software 43 (2017), pp. 268– 276.

[7]. Comon H. et al. Tree automata techniques and applications. – 2008.

[8]. Gécseg, Ferenc, and Magnus Steinby. «Tree automata» --- 1984, 2015.

[9]. Neven F., Schwentick T. Query automata over finite trees //Theoretical Computer Science. – 2002. – Т. 275. – №. 1-2. – pp. 633-674.

[10]. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology, 5(4):660–704, 2005.

[11]. G Shobha et al. – «Code clone detection– a systematic review». – B: Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2 (2021), pp. 645– 655.

[12]. D.A. Koryabkin, and V.N. Ignatyev. – «Automatic Mining of Code Fix Patterns from Code Repositories». – B: 2022 Ivannikov Memorial Workshop (IVMEM). – IEEE. 2022, – pp. 27– 34.

[13]. U.V. Tsiazhkorob и V.N. Ignatyev. – «Classification of Static Analyzer Warnings using Machine Learning Methods». – B: 2024 Ivannikov Memorial Workshop (IVMEM). – IEEE. 2024, – pp. 69– 74.

[14]. N.V. Shimchik, V.N. Ignatyev, and A.A. Belevantsev. – «Improving accuracy and completeness of source code static taint analysis». – B: 2021 Ivannikov ISPRAS OPEN Conference (ISPRAS). – IEEE. 2021, – pp. 61– 68.

Информация об авторах / Information about authors

Валерий Николаевич ИГНАТЬЕВ, кандидат физико-математических наук, старший научный сотрудник ИСП РАН, доцент кафедры системного программирования факультета ВМК МГУ. Научные интересы включают методы поиска ошибок в исходных текстах программ на основе статического анализа.

Valery Nikolayevich IGNATYEV, Cand. Sci. (Phys.-Math.) in computer sciences, senior researcher at Ivannikov Institute for System Programming RAS and associate professor at system programming division of CMC faculty of Lomonosov Moscow State University. His research interests include program analysis techniques for error detection in program source code using classical static analysis and machine learning.