ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 37 Выпуск 1

ISSN Online 2220-6426
Volume 37 Issue 1

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2025

ИСП РАН

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН
Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областям системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Редколлегия

Главный редактор - Аветисян Арутюн Ишханович, академик РАН, доктор физико-математических наук, профессор, ИСП РАН (Москва, Российская Федерация)

Заместитель главного редактора – Карпов Леонид Евгеньевич, д.т.н., ИСП РАН (Москва, Российская Федерация)

Члены редколлегии

Воронков Андрей Анатольевич, доктор физико-математических наук, профессор, Университет Манчестера (Манчестер, Великобритания)
Вирбицкайте Ирина Бонавентуровна, профессор, доктор физико-математических наук, Институт систем информатики им. академика А.П. Ершова СО РАН (Новосибирск, Россия)
Коннов Игорь Владимирович, кандидат физико-математических наук, Технический университет Вены (Вена, Австрия)
Ластовецкий Алексей Леонидович, доктор физико-математических наук, профессор, Университет Дублина (Дублин, Ирландия)
Ломазова Ирина Александровна, доктор физико-математических наук, профессор, Национальный исследовательский университет «Высшая школа экономики» (Москва, Российская Федерация)
Новиков Борис Асенович, доктор физико-математических наук, профессор, Санкт-Петербургский государственный университет (Санкт-Петербург, Россия)
Петренко Александр Федорович, доктор наук, Исследовательский институт Монреаля (Монреаль, Канада)
Черных Андрей, доктор физико-математических наук, профессор, Научно-исследовательский центр CICESE (Энсенада, Баха Калифорния, Мексика)
Шустер Ассаф, доктор физико-математических наук, профессор, Технион — Израильский технологический институтTechnion (Хайфа, Израиль)

Editorial Board

Editor-in-Chief - Arutyun I. Avetisyan, Academician of RAS, Dr. Sci. (Phys.–Math.), Professor, Ivannikov Institute for System Programming of the RAS (Moscow, Russian Federation)

Deputy Editor-in-Chief – Leonid E. Karpov, Dr. Sci. (Eng.), Ivannikov Institute for System Programming of the RAS (Moscow, Russian Federation)

Editorial Members

Igor Konnov, PhD (Phys.–Math.), Vienna University of Technology (Vienna, Austria)
Alexey Lastovetsky, Dr. Sci. (Phys.–Math.), Professor, UCD School of Computer Science and Informatics (Dublin, Ireland)
Irina A. Lomazova, Dr. Sci. (Phys.–Math.), Professor, National Research University Higher School of Economics (Moscow, Russian Federation)
Boris A. Novikov, Dr. Sci. (Phys.–Math.), Professor, St. Petersburg University (St. Petersburg, Russian Federation)
Alexandre F. Petrenko, PhD, Computer Research Institute of Montreal (Montreal, Canada)
Assaf Schuster, Ph.D., Professor, Technion - Israel Institute of Technology (Haifa, Israel)
Andrei Tchernykh, Dr. Sci., Professor, CICESE Research Centre (Ensenada, Baja California, Mexico).
Irina B. Virbitskaite, Dr. Sci. (Phys.–Math.), The A.P. Ershov Institute of Informatics Systems, Siberian Branch of the RAS (Novosibirsk, Russian Federation)
Andrey Voronkov, Dr. Sci. (Phys.–Math.), Professor, University of Manchester (Manchester, United Kingdom)

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings

© Институт Системного Программирования им. В.П. Иванникова РАН, 2025

Содержание

Table of Contents

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

CC BY

Система статического анализа для языка описания аппаратуры SystemVerilog

[1] *Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,3] *К.Н. Китаев, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>*
[1] *А.Г. Волохов, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>*
[3] *Е.В. Долгодворов, ORCID: 0000-0001-6962-6448 <dolgodvorov.ev@phystech.edu>*
[1,2,3,4] *А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *А.М. Коцыняк, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,2] *Д.О. Самоваров, ORCID: 0009-0000-2428-6654 <dmitrysamovarov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*
[3] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*
[4] *Российский экономический университет имени Г.В. Плеханова, 117997, Россия, г. Москва, Стремянный переулок, д. 36.*

Аннотация. Рост сложности современных цифровых систем и увеличение объемов кода на языках описания аппаратуры требуют эффективных инструментов для выявления ошибок на ранних этапах разработки цифровых СБИС. Для своевременного обнаружения ошибок составляются сборники правил, регламентирующие описание аппаратуры. Эти сборники содержат набор правил, описывающих неточности, ошибки и последствия их нарушения. В данной работе рассмотрен список правил, разработанный на основе опыта работы инженеров, использующих язык SystemVerilog, и представлена система статического анализа SVAN, разработанная для языка SystemVerilog и учитывающая специфику описаний аппаратуры. Предлагаемая система обеспечивает полную поддержку стандарта SystemVerilog IEEE 1800-2017 и предоставляет возможности анализа описаний на наличие структурных и семантических ошибок.

Ключевые слова: статический анализ; язык описания аппаратуры; язык описания аппаратуры Verilog; язык описания аппаратуры SystemVerilog; интегральная схема; система автоматизации проектирования (САПР); инструментарий анализа, компиляции и выполнения описаний аппаратуры slang, slang-tidy, KLEE, Yosys, Verilator, CIRCT; проект LLVM; абстрактное синтаксическое дерево АСД.

Для цитирования: Чуркин Я.А., Бучацкий Р.А., Китаев К.Н., Волохов А.Г., Долгодворов Е.В., Камкин А.С., Коцыняк А.М., Самоваров Д.О. Система статического анализа для языка описания аппаратуры SystemVerilog. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 7–40. DOI: 10.15514/ISPRAS–2025–37(1)– 1.

Благодарности: Проект финансируется Минпромторгом России в рамках ОКР «Разработка системы статического анализа для языка описания аппаратуры», шифр «САПР-Анализ» (в составе ОКР «САПР микроэлектроника», головной исполнитель – АО «МНТЦ МИЭТ»).

System for Static Analysis of SystemVerilog HDL

[1] *Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,3] *K.N. Kitaev, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>*
[1] *A.G. Volokhov, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>*
[3] *E.V. Dolgodvorov, ORCID: 0000-0001-6962-6448 <dolgodvorov.ev@phystech.edu>*
[1,2,3,4] *A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *A.M. Kotsynyak, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,2] *D.O. Samovarov, ORCID: 0009-0000-2428-6654 <dmitrysamovarov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.
[4] *Plekhanov Russian University of Economics,*
36 Stremyanny lane, Moscow, 117997, Russia.

Abstract. The growing complexity of modern digital systems and the increasing volumes of code written in hardware description languages demand effective tools for early error detection in the development of digital ASICs. To facilitate timely error detection, rule sets are created to regulate hardware descriptions. These rule sets contain a collection of rules that describe inaccuracies, errors, and the consequences of their violation. This paper discusses a list of rules developed based on the experience of engineers using the SystemVerilog language and presents the SVAN static analysis system, designed for SystemVerilog and tailored to the specifics of hardware descriptions. The proposed system provides full support for the SystemVerilog IEEE 1800-2017 standard and offers capabilities for analyzing descriptions for structural and semantic errors.

Keywords: static analysis; HDL; Verilog; SystemVerilog; IC; CAD system; analysis, compilation and simulation tools slang, slang-tidy, KLEE, Yosys, Verilator, CIRCT; LLVM project; abstract syntax tree AST.

For citation: Churkin Y.A., Buchatskiy R.A., Kitaev K.N., Volokhov A.G., Dolgodvorov E.V., Kamkin A.S., Kotsynyak A.M., Samovarov D.O. System for Static Analysis of SystemVerilog HDL. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 7-40 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-1.

Acknowledgements. The project is supported by the Ministry of Industry and Trade of the Russian Federation within the R&D project “Development of a static analysis system for a hardware description language”, code “SAPR-Analiz” (as part of the R&D project “SAPR mikroelektronika”, the main contractor is JSC “ISTC MIET”).

1. Введение

Проектирование цифровых интегральных схем является многоэтапным процессом [1], включающим спецификацию на языке описания аппаратуры (HDL, Hardware Description Language), разработку архитектуры, логический и физический синтез. Каждый этап включает верификацию, нацеленную на проверку соответствия полученного описания результату предшествующего этапа или техническому заданию, при этом исправление ошибок, обнаруженных на поздних этапах, требует значительных временных и экономических затрат. Ошибки, выявленные после внедрения изделий, могут потребовать внесения правок в проектную модель, повторного выполнения синтеза и перевыпуска микросхем. Примерами таких ошибок в больших интегральных схемах могут служить:

1. Ошибка Pentium Fdiv [2], 1994 год. При делении чисел с плавающей точкой происходила ошибка в модуле, выполняющем эту операцию. Ее причиной послужили неточности в таблице поиска, используемой при проведении операции

деления. Исправление этой ошибки требовало замены процессора и привело к убыткам в 475 миллионов долларов для компании Intel.

2. Группа уязвимостей Spectre [3], 2017 год. Ошибка в процессорах, использующих спекулятивное выполнение для получения доступа к виртуальной памяти процесса. Нейтрализация данной уязвимости требует обнаружения определенной последовательности команд в коде программ. В настоящее время для решения этой проблемы используется дополнительный микрокод или патчи операционной системы.

В данной работе рассматривается этап описания цифровой аппаратуры, на котором происходит его реализация на соответствующих языках. Описания, полученные на этом этапе, задают функциональность и структуру конечной интегральной схемы.

Одним из часто используемых языков описания цифровых схем является SystemVerilog [4]. К его преимуществам можно отнести:

1. Наличие конструкций для описания верификационных окружений (verification environment, testbench).
2. Наличие средств описания аппаратуры на различных уровнях абстракции (уровень регистровых передач и уровень логических вентилей).
3. Большое разнообразие синтаксических конструкций, ускоряющих разработку модели, например, специализированные блоки процедурной логики, такие как `always_comb`, `always_latch` и `always_ff`, интерфейсы, различные виды операторов `case` и другие.

В больших проектах наличие большого числа абстракций, предоставляемых этим языком, может привести к семантически некорректным описаниям, допустимым с точки зрения синтаксиса. При синтезе логической схемы конструкции языка преобразуются в логические вентили, а при симуляции RTL-модель вместе с тестовым окружением преобразуется в представление, пригодное для исполнения и верификации на целевом оборудовании (чаще всего в программный код на C/C++). Наиболее часто встречаются ошибки, связанные с расхождением результатов симуляции исходной RTL-модели и синтезированной логической схемы: например, различие в семантике операций 4-значной логики, частично определенные значения сигналов, различные виды присваиваний в процедурных блоках и т.д.

Для предотвращения подобных ошибок составляются сборники правил, описывающие неправильные варианты использования конструкций языка и последствия неправильного использования. Для проверки соответствия описаний списку правил может использоваться ручная или автоматическая проверка кода. Ручная проверка, выполненная экспертом, зачастую имеет более высокую точность, однако автоматическая проверка с использованием инструментов статического анализа позволяет более эффективно и быстро проверять большие проекты на соответствие списку правил. При разработке проекта должна использоваться комбинация этих техник для повышения качества проверки.

На сегодняшний день коммерческие решения для статического анализа описаний на языке SystemVerilog представлены малым количеством компаний (например, Synopsys и Cadence Design Systems), а открытые решения, хотя и находятся на стадии активного развития, на данный момент широко не представлены. В связи с этим, разработка решения статического анализа описания аппаратуры на SystemVerilog является особенно актуальной.

Данная работа посвящена разработке статических проверок описаний схем на языке SystemVerilog на соответствие разработанному списку правил, направленных на выявление и предотвращение неточностей и ошибок на ранних этапах разработки сверхбольших интегральных схем (СБИС).

2. Языки описания аппаратуры

Цифровая схема может быть описана с помощью логических вентилей и соединений между ними (gate-level netlist). Учитывая высокую сложность современных интегральных схем (10^6-10^9 вентилей), такое описание практически невозможно выполнить вручную. В связи с этим в проектировании цифровых схем получил распространение подход к описанию модели аппаратуры на уровне регистровых передач (RTL, Register-Transfer Level). На этом уровне цифровая схема описывается в терминах сигналов, регистров и логических операций между ними, не вдаваясь в подробности реализации при помощи логических вентилей. Такое описание представляет передачу значений сигналов в рамках одного периода тактового сигнала.

Языки описания аппаратуры, такие как Verilog, SystemVerilog и VHDL [5], позволяют описывать структурные и поведенческие модели на разных уровнях абстракции: на уровне регистровых передач и на уровне логических вентилей. Основным сценарием использования является проектирование RTL-модели, имитационное тестирование (simulation-based verification) RTL-модели в симуляторе и синтез по RTL-модели логической схемы. С помощью этого описания высокого уровня можно получить описание низкого уровня (gate-level netlist), которое в конечном итоге будет преобразовано в фактическую схему.

Начальными этапами синтеза HDL-описаний является построение предварительной (pre-elaborated) и развернутой (elaborated) моделей. Построение первой происходит после синтаксического разбора. Построенная предварительная модель содержит лишь описание иерархии модулей (Verilog/SystemVerilog) и сущностей (VHDL) без создания экземпляров с конкретными значениями параметров, что позволяет провести классификацию – какие модули являются верхнеуровневыми, а какие нет. Затем строится развернутая модель, в которой вычисляются все параметры всех экземпляров модулей (в том числе зависимые от значений других параметров), создаются непосредственно экземпляры модулей, а также разрешаются иерархические имена и блоки генерируемой логики (Verilog и SystemVerilog).

Языки описания аппаратуры, использующие RTL-представление, имеют синтаксис, схожий с такими процедурными языки программирования, как Pascal и C, но могут также допускать и более высокоуровневые стили описания, например, объектно-ориентированные описания в SystemVerilog. Главное отличие от языков общего назначения состоит в модели вычислений, например, существуют операции в RTL-модели схемы аппаратуры, которые должны выполняться все вместе одновременно в конце такта сигнала тактовой частоты в реальной микросхеме или в модели при симуляции несмотря на то, что они могут быть описаны в RTL-модели в императивном стиле вперемешку с последовательными операциями. Соответственно, это стоит учитывать при разработке HDL-описаний, в то время как в языках программирования все операции происходят последовательно.

В описании аппаратуры применяется аналогичный языкам программирования набор средств для объявления и использования процедур, функций и типов данных. Эти типы данных включают булевы значения, целые числа, битовые векторы и массивы фиксированной длины. Арифметические и логические операции, включая побитовые, также схожи. Основной структурной единицей [6], используемой для описания аппаратуры, является модуль (`module`), который соответствует элементам аппаратных систем. Каждый модуль имеет интерфейс, включающий входные (`input`) и выходные (`output`) сигналы. Обычно интерфейс модуля содержит специальные входные сигналы, такие как сигналы тактовой частоты (`clk`) и сброса (`reset`). Функциональная логика модуля описывается в его теле. Тело модуля может содержать фиксированное количество экземпляров других модулей (в случае иерархического описания) и фиксированное количество процессов (`process`, `always`). Процессы выполняются параллельно в рамках одного модуля. Каждый процесс может иметь список чувствительности (`sensitivity list`) и тело, содержащее последовательность операторов. Семантика некоторых операторов совпадает с семантикой аналогичных

операторов в императивных языках программирования: например, последовательные (блокирующие) присваивания, условные операторы и операторы цикла. Также используются параллельные (не блокирующие) присваивания. События в описаниях включают изменения уровня входного сигнала: приход переднего фронта сигнала (`rising edge, posedge`), заднего фронта сигнала (`falling edge, negedge`) и произвольного фронта сигнала (`event`). Важной особенностью описаний аппаратуры является специфический режим выполнения процессов: во время исполнения каждый процесс постоянно ожидает прихода события, и, если событие перечислено в списке чувствительности процесса, то выполняется соответствующее тело, и процесс вновь переходит в состояние ожидания. Присваивание значений переменным в блоках параллельных присваиваний осуществляется одновременно, а их обновленные значения становятся доступными только при следующем выполнении тела процесса.

Синтаксис описания аппаратуры на примере языка SystemVerilog приведен в листинге 1.

```
1  module SAMPLE (                          интерфейс модуля
2    input logic RST,                       входной сигнал сброса
3    input logic CLK,                       входной тактовый сигнал
4    input logic D,                         входной однобитный сигнал
5    output logic OUT                       выходной однобитный сигнал
6  );
7
8    always @(posedge CLK or negedge RST)   процесс и список чувствительности
9    begin
10     if (!RST)                            условный оператор
11       OUT <= 1'b0;                       параллельное присваивание
12     else if (D)                          условный оператор
13       OUT <= ~OUT;                       параллельное присваивание
14     else                                 условный оператор
15       OUT <= OUT;                        параллельное присваивание
16   end
17 endmodule
```

Листинг 1. Пример простого описания схемы на SystemVerilog.
Listing 1. An example of a simple circuit design in SystemVerilog.

SystemVerilog [4] – язык описания и верификации аппаратуры, являющийся расширением языка Verilog [7]. В 2005 SystemVerilog был принят как стандарт IEEE 1800-2005. В 2009 стандарт 1800-2005 был объединен со стандартом языка Verilog (IEEE 1364-2005), и была принята актуальная на тот момент версия SystemVerilog – стандарт IEEE 1800-2009. На данный момент актуальными считаются стандарты SystemVerilog IEEE 1800 редакции 2017 и 2023.

На конструкции языка SystemVerilog накладываются различные ограничения в зависимости от того, являются они частью схемы, подлежащей синтезу или нет. Синтезируемая часть описания это та, которую можно представить с помощью логических вентилей. К конструкциям синтезируемых частей описания относятся объявления портов, непрерывные присваивания, триггеры и другие. Напротив, такие конструкции как `program`, `initial` и `final` относятся к несинтезируемой части описания. Для некоторых конструкций возможен синтез, но с рядом ограничений, например, `task`.

Описание схемы должно удовлетворять определенным свойствам, которые инструмент синтеза проверяет перед его выполнением. Однако инструмент не всегда сообщает о нарушениях, обнаруженных в описании, и может исключить несинтезируемую часть из схемы, либо же некорректно осуществить синтез. Например, вызов функции может быть синтезируемым или несинтезируемым.

Конструкции верификации языка SystemVerilog позволяют проверять свойства схемы без ее непосредственной реализации в аппаратуре. С помощью симулятора можно проверить выполнение ограничений, заданных в форме темпоральных утверждений (SVA, SystemVerilog Assertions). Сценарии симуляции работы схемы описываются разработчиком с помощью этих конструкций.

Для большинства конструкций языка SystemVerilog можно статически определить, относятся они к синтезируемой части схемы или нет. Эти конструкции отличаются назначением, а также особенностями интерпретации (поведение схемы при симуляции отличается от поведения реальной схемы после синтеза).

3. Статический анализ описаний на языке SystemVerilog

Статический анализ – анализ кода, проводимый без компиляции программ (без логического синтеза аппаратуры) и их исполнения. Для статического анализа описаний аппаратуры могут быть применены техники, используемые при анализе программного обеспечения.

Рассмотрим техники, которые активно применяются в анализе описаний аппаратуры.

- **Сопоставление синтаксических конструкций**. Для использования этой техники необходим парсер языка, способный строить абстрактное синтаксическое дерево (АСД), которое представляет собой конечное помеченное ориентированное дерево, где вершины однозначно сопоставляются операторам языка описания. АСД используется для поиска паттернов кода. Поиск осуществляется с помощью специальных обходчиков дерева, собирающих данные о его вершинах. Эта техника анализа является одной из самых простых в реализации, а также самых быстрых, так как сложность выполнения обхода пропорциональна размеру дерева. Минусом такой техники является высокий процент ложных срабатываний (false positives) при проверке свойств схемы, требующих знаний о возможных значениях переменных и путях исполнения программы.
- **Анализ потока данных**. Для использования этой техники должен быть построен граф потока управления. Этот граф позволяет отслеживать зависимости по данным и управлению в схеме. Вершины этого графа представляют собой блоки последовательного выполнения операторов в описании. Операторы внутри этих блоков могут порождать факты, которые исследуются в ходе анализа, или говорить о недостоверности факта при выполнении конкретного оператора. В ходе анализа потока данных информация о фактах относительно операторов описания распространяется по всем возможным путям исполнения тестового описания или распространяется на возможные состояния схемы. Распространение происходит итерационно до момента полного совпадения состояний на соседних итерациях. Этот метод анализа позволяет решать такие задачи как: нахождение мертвого кода, анализ живых переменных. Зачастую такой анализ используется для оптимизации вычислений в процессах или для удаления частей схемы, которые не влияют на ее выходы.
- **Символьное исполнение (симуляция)**. Техника позволяет симулировать схему без заданных значений входных сигналов. Инструмент символьного исполнения анализирует возможные пути выполнения для данной схемы и для каждого такта тактового сигнала генерирует набор входных данных для обхода всех путей. Обычно такой запуск может работать лишь ограниченное число тактов, так как символьное исполнение порождает экспоненциальное число возможных путей. При обнаружении пути, на котором есть нарушение определенного свойства схемы, путь достижения запоминается и выдается инструментом символьной симуляции. Минусом данного метода является большая затрата ресурсов на выполнение симуляции множества путей. На данный момент есть несколько работ [8-9], в

которых используются техника символьного исполнения программ, полученных трансляцией SystemVerilog-описания в описание на языке программирования общего назначения. С помощью этого метода можно эффективно решать такие задачи, как обнаружение выхода за границы массива, выявление множественного присваивания одной переменной и другие. Однако на данный момент нет открытых инструментов, реализующих символьное исполнение с полной поддержкой описаний на языке SystemVerilog.

Описанные методы статического анализа покрывают значительную часть ошибок, которые обрабатываются статическим анализатором.

3.1 Обзор инструментов статического анализа описаний на языке SystemVerilog

Для определения методов реализации детекторов был проведен обзор доступных статических анализаторов описаний на SystemVerilog. В настоящее время имеется множество инструментов различного назначения, обладающих функциональностью для проведения статического анализа описаний на языке SystemVerilog, однако в большинстве своем эти инструменты, предоставляются по коммерческим лицензиям, но есть небольшая часть с открытым исходным кодом.

В данной главе представлен обзор существующих открытых и коммерческих решений, включая линтеры, парсеры, симуляторы, инструменты предварительного синтеза.

Кроме того, в данной главе произведен анализ соответствия открытых инструментов ряду критериев. Инструменты оценены с точки зрения поддержки стандарта языка SystemVerilog, наличия необходимых для анализа представлений, лицензирования, функциональных возможностей, таких как интерфейс для реализации статических детекторов, а также интеграции с другими инструментами. Поддержка языка оценивается на основании результатов запуска парсеров на синтетическом наборе тестов sv-tests [10], а также с учетом документации к инструментам. Интеграция рассматривается как возможность совместного использования инструментов для выполнения статического анализа, что в свою очередь может повысить эффективность процедур оценки и улучшить качество анализируемого кода.

3.1.1 Коммерческие инструменты для анализа описаний на языке SystemVerilog

Существует ряд коммерческих инструментов, позволяющих выявить ошибки описаний дизайна цифровой аппаратуры на языках Verilog, SystemVerilog. Наиболее популярными являются SpyGlass, VCS, JasperGold и Xcelium, краткий обзор которых представлен ниже. Стоит отметить, что синтаксические и семантические проверки свойств описаний дизайнов аппаратуры в разной степени присутствуют во всех коммерческих инструментах, в том числе и в тех, которые не представлены в обзоре (Vivado от компании Xilinx, Questa от компании Siemens и другие).

Spyglass [11] – это инструмент статической верификации и анализа RTL-кода, разработанный компанией Synopsys. Инструмент содержит модуль Lint, выполняющий статический анализ кода, проверяя его на соответствие определенным правилам проектирования и выявляя потенциальные ошибки при проектировании. Также в инструменте представлены модули CDC, RDC и Power, предназначенные для выявления ошибок пересечения доменов тактовой частоты, некорректного использования асинхронных сбросов и анализа доменов питания соответственно. Полный список проверок, реализуемых SpyGlass, не опубликован в открытом доступе, но по заявлениям разработчиков покрывает основные ошибки, возникающие в ходе проектирования и разработки описаний схем. Полнота поддержки стандартов не указана явно, но инструмент предназначен для работы с описаниями на SystemVerilog, что подразумевает соответствие стандартам IEEE 1800.

VCS (Verilog Compiled Simulator) [12] представляет собой высокопроизводительный симулятор и инструмент для анализа описаний на языках Verilog и SystemVerilog, разработанный компанией Synopsys. VCS практически полностью поддерживает актуальные стандарты Verilog и SystemVerilog (заявлена поддержка IEEE 1800-2023), осуществляет проверку набора правил линтинга и предлагает широкий спектр средств отладки симулируемой модели, включая визуализацию, трассировку и анализ значений сигналов. Также инструмент поддерживает верификацию по методологии UVM (Universal Verification Methodology) [13].

JasperGold [14] – это инструмент формальной верификации, разработанный компанией Jasper Design Automation (сейчас принадлежит Cadence Design Systems). Он обеспечивает формальную верификацию описаний на языках Verilog и SystemVerilog, проводит доказательства свойств и проверки характеристик цифровых схем на соответствие заявленным спецификациям без использования симуляции. JasperGold также позволяет разрабатывать и добавлять модульные проверки, которые можно комбинировать для создания комплексных сценариев верификации.

Xcelium [15], разработанный компанией Cadence, является аналогом VCS. Инструмент практически полностью поддерживает актуальные стандарты Verilog и SystemVerilog, осуществляет статические проверки правил линтинга, предоставляет возможности для многопоточной симуляции, поддерживает верификацию по методологии UVM и обладает графическим интерфейсом (в отличие от VCS).

3.1.2 Инструменты с открытым исходным кодом для анализа описаний на языке SystemVerilog

В области разработки и анализа описаний аппаратуры на языке SystemVerilog существует ряд инструментов с открытым исходным кодом, каждый из которых предоставляет специфические возможности для синтаксического анализа, линтинга и симуляции кода. В данной работе рассматриваются и сравниваются шесть таких инструментов: Verible, svlint, Verilator, Surelog, CIRCT и slang. Популярные инструменты Yosys [16] и Icarus Verilog [17] в обзоре не фигурируют, так как изначально создавались для логического синтеза и симуляции описаний цифровой аппаратуры на Verilog и не являются инструментами статического анализа.

Verible [18] представляет собой набор инструментов для работы с описаниями на языке SystemVerilog, включая средства статического анализа и линтинга. Особенностью Verible является построение конкретного синтаксического дерева (дерева разбора) для HDL-описания до запуска препроцессора. На текущем этапе разработки Verible частично поддерживает стандарт SystemVerilog, и некоторые конструкции языка могут быть не реализованы. Инструмент предоставляет собственную инфраструктуру для реализации новых детекторов и матчеров – функций для поиска определенных паттернов в дереве разбора. На данный момент реализованы линтер и языковой сервер. К преимуществам Verible также можно отнести свободную лицензию Apache 2.0, которая способствует его широкому использованию и развитию.

svlint [19] – это инструмент линтинга, предназначенный для анализа описаний на языке SystemVerilog. Он использует sv-parser [20] в качестве парсера, который практически полностью покрывает стандарт SystemVerilog. svlint выполняет анализ на основе дерева разбора, построенного sv-parser; при этом, в отличие от Verible, инструмент выполняет препроцессирование описания. svlint предоставляет инфраструктуру для реализации собственных детекторов и матчеров синтаксических конструкций. Кроме того, инструмент включает собственный языковой сервер svls, реализующий протокол LSP. Распространяется под лицензией MIT, что обеспечивает его свободное использование и модификацию.

Verilator [21] – это инструмент симуляции описаний аппаратуры на языках Verilog и SystemVerilog, разрабатываемый CHIPS Alliance. Он частично поддерживает конструкции стандарта IEEE 1800-2017 языка SystemVerilog и обладает несколькими внутренними представлениями описаний аппаратуры: АСД (включая XML-представление АСД), представление в виде кода на языках C++ или SystemC для эмуляции аппаратуры, а также представление DfgGraph на основе графа потока данных для оптимизатора комбинационной логики. В частности, в работе [9] эмуляция аппаратуры в виде C++ кода использовалась для символьной симуляции схем при интеграции с инструментом символьного выполнения KLEE [22]. Verilator предоставляет инфраструктуру для реализации собственных детекторов, включая обходчики АСД. Встроенный в инструмент модуль линтига анализирует и находит семантические ошибки, которые могут повлиять на результаты симуляции или синтеза. Инструмент лицензирован под GNU Lesser General Public License v3.0 (LGPL-3.0), что ограничивает его коммерческое использование.

Surelog [23] – это компилятор описаний аппаратуры на языке SystemVerilog, который на выходе строит модель аппаратуры в специальном формате (Universal Hardware Data Model, UHDM [24]). UHDM является открытым форматом, разработанным CHIPS Alliance для представления RTL-модели в универсальном формате, подходящем для интеграции парсера Surelog с различными инструментами синтеза и симуляции, например, Yosys и Verilator (для которых существуют трансляторы из UHDM в их внутренние представления). С помощью средств UHDM можно описывать как предварительную, так и развернутую RTL-модель. Парсер Surelog практически полностью поддерживает стандарт IEEE 1800-2017. Инструмент строит АСД, имеет встроенный препроцессор и генерирует развернутую модель UHDM, что позволяет выполнять межмодульный анализ. Surelog предоставляет Python API для создания правил линтинга и обхода АСД, однако не имеет C++ API для реализации статических детекторов. Лицензируется под Apache 2.0.

CIRCT [25] – представляет собой набор инструментов для симуляции, линтинга и компиляции описаний на языках описания аппаратуры, таких как Chisel, SystemC и Verilog. Основной целью CIRCT является представление моделей, описанных на различных языках в виде промежуточного представления MLIR [26], соответствующего форме единственного присваивания (SSA). MLIR (Multi-Level Intermediate Representation) – часть проекта LLVM, предоставляющая инфраструктуру для создания и работы с промежуточными представлениями. Она включает операции, регионы и блоки, образующие рекурсивную структуру. Диалекты объединяют операции, типы и атрибуты для конкретных задач. MLIR поддерживает более 40 встроенных диалектов и позволяет создавать пользовательские с помощью специализированного языка.

CIRCT предоставляет диалекты, относящиеся к различным языкам описания аппаратуры, которые могут быть преобразованы в основные (core) диалекты (HW, Comb, Seq, Interop) или же могут быть подвергнуты анализу в рамках конкретного диалекта. CIRCT содержит как промежуточные представления, так и различные вспомогательные абстракции, такие как граф инстанциации модулей, что позволяет проводить межмодульный анализ. Поскольку CIRCT основывается на инфраструктуре MLIR, в настоящее время проводятся работы по эмуляции аппаратуры с помощью трансляции описаний в LLVM. На данный момент активно осуществляется интеграция инструмента slang в качестве парсера для SystemVerilog в рамках CIRCT. Следует отметить, что данный инструмент считается экспериментальным и пока не используется широко с другими инструментами, однако у него есть активное сообщество разработчиков. Лицензия инструмента Apache 2.0.

slang [27] – это инструмент для статического анализа и компиляции описаний аппаратуры на языке SystemVerilog, обладающий практически полным покрытием стандарта IEEE 1800-2017, а также стандарта IEEE 1800-2023. slang строит АСД и предоставляет инфраструктуру для реализации статических детекторов, включая интерфейс для обхода АСД. Помимо

компилятора и статического анализатора инфраструктура slang включает несколько дополнительных инструментов:

- Rewriter: инструмент, позволяющий автоматически вносить изменения в файлы описаний аппаратуры посредством модификации АСД.
- Netlist: инструмент для выполнения структурных проверок, строящий связи и соединения между различными модулями.
- slang-tidy [28]: инструмент статического анализа описаний на языках SystemVerilog и Verilog, использующих сопоставление синтаксических конструкций.

slang активно используется в других проектах в качестве фронтенда, например, в проекте CIRCT и в утилите Yosys+slang [29]. Инструмент распространяется под лицензией MIT, что способствует его интеграции и расширению в рамках других решений.

3.1.3 Сравнение инструментов с открытым исходным кодом

В табл. 1 представлено сравнение инструментов с открытым исходным кодом с точки зрения возможностей статического анализа SystemVerilog.

Табл. 1. Сравнение анализаторов с открытым кодом.
Table 1. Open-source analyzers comparison results.

Инструмент	**Поддержка SystemVerilog**	**Внутренние представления**	**Виды анализа**	**Интерфейс для создания детекторов**
Verible	частичная	Дерево разбора	Сопоставление синтаксических конструкций	+
svlint	практически полная	Дерево разбора	Сопоставление синтаксических конструкций	+
Verilator	частичная	АСД, C++, SystemC, граф потока данных	Сопоставление синтаксических конструкций; анализ потока данных	–
Surelog	практически полная	АСД, UHDM	Сопоставление синтаксических конструкций	+ (Python)
CIRCT	частичная	MLIR	Анализ потока данных на SSA	–
slang	практически полная	Дерево разбора, АСД	Сопоставление синтаксических конструкций	+

Как видно из таблицы, практически полную поддержку стандарта IEEE 1800-2017 предоставляют только такие инструменты, как svlint, Surelog и slang.

Инструменты Verible и svlint не позволяют проводить статический анализ с помощью сопоставления синтаксических конструкций, так как не способны построить АСД по описанию аппаратуры. Важно учитывать возможность расширения функциональности статического анализатора путем добавления других форм анализа или его интеграции с инструментами, поддерживающими такие методы.

Verilator является достаточно удобным и мощным инструментом для статического анализа с точки зрения инфраструктуры. Анализ производится в нем поэтапно в процессе трансляции

и построения модели на C++. Но данный инструмент создавался в первую очередь для языка Verilog, соответственно, поддержка стандарта языка SystemVerilog на данный момент достаточно ограничена. Также, немаловажным аспектом, повлиявшим на исключение этого инструмента из списка рассматриваемых, стала ограничивающая лицензия.

При выборе инструмента для реализации статического анализа описаний на SystemVerilog важным фактором стало наличие интерфейса для создания статических детекторов и интерфейса для выдачи диагностических предупреждений. В данном контексте наиболее подходящим решением оказался инструмент slang, так как он предоставляет широкофункциональные интерфейсы для разработки статических детекторов (через отдельный инструмент статического анализа – slang-tidy), а также интерфейсы для создания и выдачи диагностических предупреждений.

Также slang активно развивается сообществом, практически полностью поддерживает стандарт IEEE 1800-2017, демонстрирует широкую функциональность для выполнения углубленного анализа описаний и интегрирован в другие средства в качестве фронтенда, что обеспечивает возможность взаимодействия в экосистеме открытых инструментов для анализа описания аппаратуры.

С учетом перечисленных преимуществ, именно slang был выбран в качестве основной инфраструктуры для реализации системы статического анализа.

4. Правила проверки описаний на языке SystemVerilog

Несмотря на мощность и гибкость языка описания и верификации аппаратуры SystemVerilog, отсутствие четких руководящих принципов и стандартов проектирования может привести к разнообразным проблемам при разработке, таким как снижение читаемости кода, увеличение количества ошибок и усложнение процесса симуляции (моделирования), верификации и синтеза.

В этой главе рассматривается процесс и мотивация создания списка правил, разработанного на основе опыта инженеров российских дизайн-центров электроники и анализа зарубежных коммерческих анализаторов описаний на языке SystemVerilog. Составленный список направлен на унификацию подходов к написанию кода и повышение общего качества проектов и содержит как правила из стандартов (например, STARC [30]), так и пользовательские правила, отражающие опыт использования коммерческих инструментов.

Разработка правил, направленных на выявление и предотвращение неточностей и ошибок на ранних этапах разработки, значительно сокращает риск возникновения проблем в последующих этапах, что ведет к более надежным и устойчивым проектам с точки зрения их разработки и сопровождения. Стандартизация подходов к написанию кода способствует повышению совместимости кода между различными инструментами и средами разработки. Это особенно важно в условиях быстро меняющейся технологической базы и разнообразия используемых САПР.

В процессе сбора и анализа опыта инженеров, работающих с Verilog и SystemVerilog, были выявлены наиболее частые проблемы и ошибки, возникающие в процессе проектирования, реализации, верификации и синтеза проектов. Для каждой проблемы было сформулировано конкретное правило, направленное на ее предотвращение и соответствующее улучшение качества кода. Также были проанализированы правила из таких анализаторов, как SpyGlass и VCS, реализующие известные стандарты кодирования (например, OpenMore [31], DO-254 [32] и STARC), с целью уточнения сформулированных ранее правил, используя опыт коммерческих решений анализа описаний схем. Итоговый список правил был классифицирован по **18** категориям, соответствующим различным аспектам языка и процесса разработки. Такая структура позволяет легко ориентироваться в правилах и применять их к соответствующим областям кода. Полная классификация правил по категориям представлена в табл. 2.

Табл. 2. Классификация выделенных правил.
Table 2. Rules classification.

Категория	**Описание**
arith	проверки арифметических операций
assign	проверки операций присваивания
case	проверки операторов case
design	проверки логики и семантики описания схем
fsm	проверки конечных автоматов в дизайне
latch	проверки использования защелок
loop	проверки операторов цикла
port	проверки использования портов
pp	проверки использования директив препроцессора
range	проверки выходов за границы
reset	проверки конфликтов, связанных с сигналами сброса
signal	проверки корректности использования сигналов
synth	проверки синтезируемости описания схем
style	проверки оформления описания схем
timing	проверки конфликтов, связанных с временными задержками
type	проверки конфликтов, связанных с типами
variable	проверки использования переменных
width	проверки конфликтов, связанных с размером битовых векторов

Правила попадающие в определенную категорию, обладают общей характеристикой: например, правила категории `synth` направлены на проверку корректности синтезируемости описания аппаратуры с точки зрения определенных конструкций или семантических свойств (например, к данной категории относится правило, когда выходной регистр модуля обновляется на разных фазах сигнала тактовой частоты), правила из категории `signal` проверяют корректность описаний на SystemVerilog при работе с сигналами (например, к данной категории относится правило, когда используется сигнал, которому никогда не устанавливается значение).

При разработке правил учитывались:

- Соответствие стандартам: правила должны соответствовать официальным спецификациям языка SystemVerilog и общепринятым практикам.
- Практическая применимость: правила должны быть применимы к реальному коду и не создавать чрезмерную нагрузку на разработчиков.
- Ясность и однозначность: формулировки правил должны быть понятными и не допускать неоднозначных трактовок.

В результате был сформулирован список из **113** правил для языков описания аппаратуры Verilog и SystemVerilog. Для реализации статических детекторов для этих правил была разработана система для статического анализа SVAN, описание которой дано в следующем разделе.

Отметим, что отдельно были выделены правила, связанные с корректностью синхронизации (CDC и RDC) и корректностью использования доменов питания, однако рассмотрение этих правил выходит за рамки работы.

5. Система статического анализа SystemVerilog

На рис. 1 представлена структурная схема разработанной системы статического анализа описаний аппаратуры на языке SystemVerilog – SVAN.

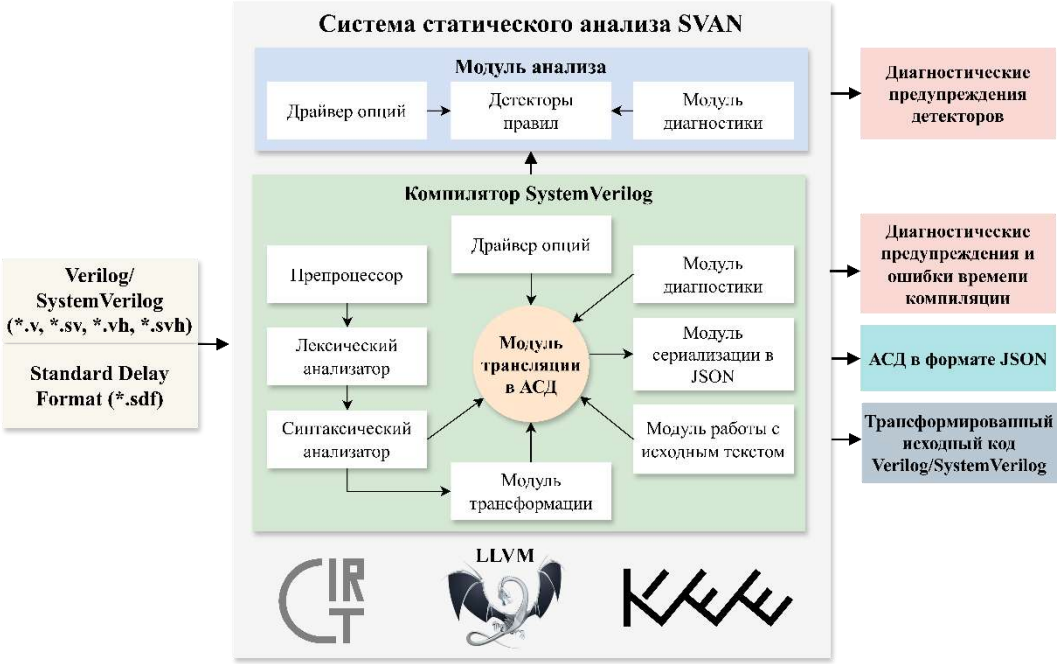


Рис. 1. Структурная схема системы статического анализа SVAN.
Fig. 1. Structural diagram of the SVAN static analysis system.

На верхнем уровне система статического анализа состоит из следующих компонентов:

- компилятор SystemVerilog;
- модуль анализа;
- подсистема анализа на основе инфраструктур CIRCT, LLVM и KLEE.

На вход системе поступают файлы с исходным текстом описания аппаратуры на языках Verilog или SystemVerilog, файлы со спецификацией задержек в формате SDF (Standart Delay Format), а также опции для компилятора и модуля анализа.

Модуль работы с исходным текстом анализирует пути к директориям с файлами и библиотеками с исходным кодом описаний аппаратуры на языках Verilog и SystemVerilog и составляет один или несколько единиц трансляции, в зависимости от переданных опций.

Затем файлы с описанием аппаратуры в виде одной или нескольких единиц трансляции обрабатываются препроцессором, который отвечает за раскрытие макроопределений в зависимости от переданных опций, а также за разрешение зависимостей включаемых файлов и подключаемых библиотек. Предобработанные единицы трансляции разбиваются на лексемы лексическим анализатором для их дальнейшей синтаксически управляемой трансляцией методом рекурсивного спуска, которая реализована в модуле синтаксического анализатора. В процессе синтаксически управляемой трансляции для каждой единицы

трансляции строится синтаксическое дерево разбора, по которому, затем, строится абстрактное синтаксическое дерево модулем трансляции в АСД.

Синтаксическое дерево разбора единицы трансляции можно трансформировать с помощью программного интерфейса модуля трансформации, который позволяет описывать обходчики синтаксического дерева и преобразовывать его узлы. Модуль также предоставляет возможность трансляции преобразованного синтаксического дерева разбора в Verilog и SystemVerilog, а также генерации АСД для дерева разбора с помощью модуля трансляции в АСД.

Модуль трансляции АСД рекурсивным обходом в глубину преобразует узлы синтаксического дерева разбора в узлы АСД, в результате строится предварительная модель каждой единицы трансляции. Затем по предварительной модели с помощью дополнительного семантического анализа зависимостей данных строится развернутая модель для каждой единицы трансляции. Ошибки и предупреждения, собранные в процессе лексического, синтаксического и семантического анализа обрабатываются и выдаются в виде предупреждений в файл или стандартный поток вывода модулем диагностики.

Построенное АСД подается на вход модулю анализа, который запускает наборы детекторов правил по категориям, специфицированным в конфигурационном файле. Детекторы в процессе обхода и анализа АСД выдают диагностические предупреждения с привязкой к позиции в файлах с исходным кодом. Предупреждения обрабатываются и выводятся в файл или стандартный поток вывода модулем диагностики.

АСД можно сохранить для обработки сторонними инструментами с помощью модуля сериализации в JSON, который получает на вход построенный АСД от модуля трансляции в АСД, затем рекурсивным обходом в глубину преобразует его узлы вместе с атрибутами в формат JSON, а также выводит его в файл или в стандартный поток вывода в зависимости от переданной опции.

5.1 Компилятор SystemVerilog

Компилятор основан на программной библиотеке с открытым исходным кодом slang [27], которая предоставляет различные компоненты для лексического анализа, синтаксического анализа, проверки типов и элаборации описаний на SystemVerilog. Библиотека slang предоставляет инструментарий для трансляции и анализа проектов на SystemVerilog, а также API для использования в качестве внешнего интерфейса для инструментов синтеза, симуляторов, линтеров, редакторов кода и инструментов рефакторинга.

Компилятор решает задачи, связанные с трансляцией описаний на языке SystemVerilog: непосредственно трансляцию текстового описания (препроцессирование, лексический и синтаксический анализ); построение дерева разбора; построение абстрактного синтаксического дерева; построение предварительной модели; построение развернутой модели.

Синтаксический анализ реализован методом рекурсивного спуска, который обеспечивает устойчивость к ошибкам в исходном тексте. Этот подход позволяет продолжить анализ даже при возникновении синтаксических ошибок. Результатом работы синтаксического анализатора является построенное дерево разбора, состоящее из синтаксических узлов разных типов. АСД находится на более высоком уровне абстракции и строится на отдельном этапе.

Построение дерева разбора, абстрактного синтаксического дерева, предварительной и развернутой модели реализовано с помощью модуля трансляции в АСД.

В процессе компиляции строится развернутая модель описания путем элаборации предварительной модели. Эта модель представлена в виде АСД. В процессе элаборации происходит, в том числе создание экземпляров модулей, соединение портов, подстановка параметров. В каждом проекте на языке SystemVerilog есть модуль верхнего уровня,

описывающий схему целиком, в то время как экземпляры остальных модулей создаются внутри модуля верхнего уровня. Такое представление задает иерархию соединения модулей итоговой схемы. Создание экземпляров модулей позволяет вычислять значения параметров для каждого экземпляра, и, как следствие, провести межмодульный анализ.

В рамках данной работы парсер slang был доработан в целях обеспечения полной поддержки соответствия стандарту SystemVerilog IEEE 1800-2017, в частности:

- Реализована поддержка статических проверок определений и разрешений множественных сигналов тактовой частоты (разделы стандарта 16.13 и 16.16);
- Реализована поддержка статической проверки невырожденности последовательностей (раздел стандарта 16.12.22) и доработка механизма проверок последовательностей, возвращающих только пустые сопоставления;
- Добавлен парсер файлов со спецификацией задержек в формате SDF и реализован механизм сопоставления и аннотации соответствующих конструкций, описывающих задержки и временные ограничения в SystemVerilog (раздел стандарта 32);
- Реализована механизм статической проверки покрытия строк в таблице состояний всех возможных состояний чувствительного к фазе последовательностного UDP (раздел стандарта 29.6), а также доработан механизм проверок перекрытия строк таблицы состояний UDP;
- Реализован механизм статической проверки перекрытия псевдонимов (алиасов) сигналов (раздел стандарта 10.11);
- Доработан механизм статических проверок виртуальных интерфейсов (раздел стандарта 25.9);
- Доработан механизм проверки ограничений на использование методов последовательностей `triggered` и `matched` в SystemVerilog;
- Другие доработки и исправления, связанные с лексическим и синтаксическим анализатором, PLA, рекурсивными свойствами, арифметикой целых чисел, отложенными операторами `assert` и т.д.

5.2 Модуль анализа

Модуль анализа основан на инструменте статического анализа описаний на языках SystemVerilog и Verilog slang-tidy из библиотеки slang. Инструмент предназначен для статического анализа описаний на языках SystemVerilog и Verilog, в том числе для реализации статических детекторов, использующих сопоставление синтаксических конструкций с помощью механизма обходчиков АСД.

Все детекторы этого инструмента реализуют общий интерфейс – `TidyCheck`, используемый для регистрации и вывода диагностики. Интерфейс предоставляет метод `check`, который принимает в качестве входного параметра корень АСД и является входной точкой для запуска одного или нескольких обходчиков, анализирующих АСД. Также интерфейс предоставляет методы для описания информации о детекторе, такие как название детектора, короткое описание, уникальный код диагностики и другие (листинг 2).

Большая часть детекторов требует реализации анализа с помощью обхода синтаксического дерева. Для этой цели в библиотеке slang предусмотрен интерфейс обходчика АСД. Обход производится от корневой вершины, которой обычно является вершина с типом – «единица компиляции» (`CompilationUnit`) к листовым. Пользователь может определить обработчик (`handler`) для каждого типа вершины АСД и собирать необходимую информацию во время обхода. Стоит отметить, что каждый узел АСД компилятора SVAN наследует один из четырех базовых типов: утверждение (`Statement`), выражение (`Expression`), символ

(`Symbol`) или тип (`Type`), подобно тому, как это реализовано в инфраструктуре компилятора Clang [33]. При определении метода обработчика определенного типа узла АСД этот обработчик вызывается обходчиком после выбора наиболее подходящего варианта с точки зрения иерархии типов.

```
class TidyCheck {
public:
    virtual bool check(const slang::ast::RootSymbol& root) = 0; // Точка входа анализа

    virtual std::string name() const = 0;              // Наименование детектора
    virtual std::string description () const = 0;      // Описание
    virtual std::string shortDescription() const = 0; // Краткое описание

    virtual slang::DiagCode diagCode() const = 0; // Целочисленный уникальный код
                                                  // диагностики, выдаваемый детектором
    virtual slang::DiagnosticSeverity diagSeverity() const = 0;
    virtual std::string diagString() const = 0;  // Строковое представление
                                                 // диагностического предупреждения
protected:
    slang::Diagnostics diagnostics;
    slang::TidyKind kind;
}
```

Листинг 2. Интерфейс класса TidyCheck.
Listing 2. TidyCheck interface.

Изначально инструмент slang не поддерживал автоматический обход дочерних узлов, что существенно усложняло логику кода детекторов. Интерфейс детекторов был доработан для поддержки автоматического определения узлов-потомков и их обхода.

В некоторых случаях при реализации детекторов правил может потребоваться классифицировать и отфильтровать отдельные файлы или участки кода на Verilog и SystemVerilog, которые относятся к тестовой среде (testbench), чтобы они не попадали под проверку свойств синтезируемого описания. Для этого реализован механизм указания пользователем того набора файлов и конструкций языка, которые необходимо отбросить при анализе.

Для упрощения разработки детекторов, анализирующих свойства тактовых сигналов и сигналов сброса, разработаны эвристические алгоритмы поиска таких сигналов в модулях, основанные на анализе их использований в процедурных блоках. Кроме того, поддерживается возможность пометки тактовых сигналов и сигналов сброса по именам через опции slang-tidy. Среди других добавленных возможностей стоит упомянуть эвристический механизм анализа и распространения значений сигналов в многозначных логиках, а также механизм определения комбинационной и последовательностной логики в процедурных блоках.

5.3 Метод анализа описаний с использованием инфраструктуры CIRCT и KLEE

Детекторы, для которых требуется анализ на уровне графа потока управления или графа потока данных, могут быть реализованы с использованием инфраструктуры CIRCT, возможности которой в совокупности покрывают виды анализа, сложно реализуемые с достаточной степенью точности на уровне структурного анализа на АСД. Примером такого анализа может служить распространение сигналов, содержащих `X и `Z биты.

Также существует возможность реализации детекторов, требующих статический анализ с использованием механизма символьного выполнения. Такой механизм реализован в инструменте с открытым исходным кодом KLEE [22]. KLEE является инструментом

динамического символьного выполнения для анализа всех возможных путей выполнения программы, построенным на базе LLVM. Инструмент заменяет конкретные входные данные символьными переменными, что позволяет исследовать множество путей выполнения, которые могли бы возникнуть при различных комбинациях входных данных. В ходе символьного выполнения KLEE осуществляет проверку состояний программы на наличие ошибок, включая переполнения, деление на ноль и нарушения свойств безопасности.

На рис. 2 представлена предварительная схема возможных путей трансляции SystemVerilog во внутреннее представление LLVM IR [34], на основе которого осуществляется символьное исполнение в KLEE.

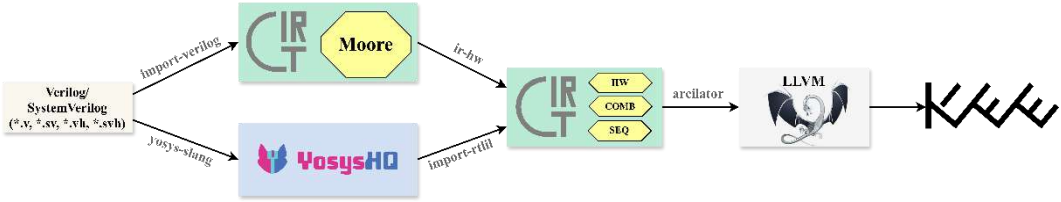


Рис. 2. Схема возможных путей трансляции в LLVM IR.
Fig. 2. LLVM IR translation paths.

Один из путей подразумевает трансляцию АСД представления описания, полученного Slang, в промежуточное представление CIRCT, используя диалект Moore [35]. Полученное представление в диалекте Moore преобразуется в представление на множестве основных низкоуровневых диалектов CIRCT – таких, как Comb, Seq, HW, – или других, подходящих для моделирования аппаратуры и оптимизации. После этого, с использованием симулятора arcilator [36], разработанного в рамках проекта CIRCT, выполняется трансляция в LLVM IR. Данный путь не предполагает использования промежуточных инструментов для получения LLVM IR, что является существенным преимуществом.

Второй путь подразумевает трансляцию АСД-представления в промежуточное представление Yosys RTLIL [37] при помощи расширения для инструмента Yosys (yosys-slang [29]) с дальнейшей конвертацией в CIRCT и LLVM IR. Данный путь предполагает использование дополнительного инструмента, но предоставляет возможность для анализа описаний с помощью средств Yosys, например, обнаружение защелок и выделение и анализ детерминированных конечных автоматов (FSM, Finite State Machine).

После трансляции описания аппаратуры на SystemVerilog в LLVM IR полученное представление передается на вход инструменту KLEE для символьного выполнения.

Выбор окончательного набора инструментов для реализации детекторов проверки правил, требующих анализ потока данных и механизма символьного выполнения, будет произведен в рамках дальнейшего исследования.

6. Разработка и реализация детекторов

В данном разделе рассмотрена реализация двух детекторов. За основу реализации детекторов для разработанных правил была выбрана инфраструктура анализатора slang-tidy, интерфейс и основные принципы работы которого были разобраны в разделе 5.

6.1 Детектор правила PORT10

Рассмотрим реализацию простого детектора для проверки правила PORT10, которое звучит следующим образом:

> «Шины инстанциаций модуля не должны быть присоединены в обратном порядке»

Использование обратных подключений (например: `[15:0]->[0:15]`) является синтаксически корректной, но плохой практикой процесса разработки, так как может

приводить к ошибкам чтения неинициализированных битов.

В листинге 3 приведен пример описания на языке SystemVerilog с конструкциями как нарушающими данное правило, так и соответствующими ему.

```
module test (input clk, input[0:5] in, inout [1:0] io);
endmodule


module top (input clk);
  wire clk;
  wire [0:5] sig1, sig2;
  wire [5:0] sig3, sig4;
  test t1(clk, sig3, sig4[1:0]);   // Нарушение – порядок битов sig3
                                   // не совпадает с in
  test t2(clk, sig1, sig2[0:1]);   // Нарушение – порядок битов sig2
                                   // не совпадает с io
  test t3(clk, sig1, sig3[1:0]);   // Нет нарушения
endmodule
```

*Листинг 3. Иллюстрация р*азличных аспектов правила PORT10.
Listing 3. Various aspects of the PORT10 rule.

Рассмотрим возможную реализацию детектора для данного правила с использованием slang-tidy. Для реализации данного правила достаточно анализа АСД представления для поиска узлов с типом «инстанциация модуля» (`InstanceSymbol`).

В разделе 5.2 был описан интерфейс детекторов slang-tidy. Реализация метода `check` и обходчика `InstanceVisitor` для данного правила представлена в листинге 4.

```
1  bool InstanceVisitor::handle(const InstanceSymbol& instanceSymbol) {
2   for (auto pConn : instanceSymbol.getPortConnections()) {
3     const auto* conn = pConn->getExpression(); // Извлечение подключаемого к порту сигнала
4     const auto* port = &pConn->port;           // Извлечение символа порта
5     const auto& cType = conn->getType();
6     const auto& pType = port->getType();
7     // Проверка, что типы являются упакованными массивами
8     if (!cType.isPackedArray() || !pType.isPackedArray())
9       continue;
10    // Проверка, что размер упакованных массивов больше 1.
11    // В этом случае не нужно выдавать предупреждение
12    if (pType.range.left == pType.range.right || cType.range.left == cType.range.right)
13      continue;
14
15    // Если у подключаемого сигнала левая граница меньше (больше) правой,
16    // а в определении порта левая граница больше (меньше) правой, то выдать диагностику.
17    if ((pType.range.left > pType.range.right) != (cType.range.left > cType.range.right))
18      diag(conn->location, "Шины присоединены в обратном порядке");
19  }
20  return true; // Обход дочерних узлов по умолчанию
21 }
22
23 // Регистрация детектора
24 class RevConnBuses : public TidyCheck {
25 public:
26    bool check(const RootSymbol& root) override {
27      // Вызов основного обходчика
28      InstanceVisitor visitor(diagnostics, root);
29      root.visit(visitor);
30      return diagnostics.empty();
31    }
32 };
```

Листинг 4. Реализация обходчика инициализаций модулей детектора правила PORT10.
Listing 4. Implementation of the PORT10 rule module initialization detector visitor.

Листинг описывает следующие шаги анализа:

1. Обходчик `InstanceVisitor` посещает все узлы АСД, начиная с корня, пока не встретит узел с типом `InstanceSymbol`, для которого описан обработчик в методе `handle` (строки 1–21).
2. Обработчик в цикле перебирает все соединения (строка 2).
3. Обработчик извлекает из соединений подсоединяемый сигнал (строка 3) и определение порта (строка 4), к которому этот сигнал подключается.
4. После извлечения типов соединения и порта (строки 5–6), происходит проверка, что они являются упакованными массивами (строка 8) (другими словами – проверяется, что их можно рассматривать, как единые сигналы, а не массивы сигналов).
5. Фильтруются случаи (строка 12), когда диапазоны длин порта и подключаемого сигнала больше 1; в противном случае прямой и обратный порядок подключения битов совпадает.
6. Происходит итоговая проверка (строки 17–18) того, что порядок подключения битов совпадает; в ином случае выдается предупреждение.

Стоит отметить, что приведенная реализация детектора правила PORT10 представлена в упрощенном виде для репрезентативности, реализация детектора в модуле анализа несущественно отличается.

6.2 Детектор правила ASSIGN04

Рассмотрим реализацию более сложного детектора для проверки правила ASSIGN04, которое звучит следующим образом:

> «Неблокирующее присваивание не должно использоваться в комбинационной логике, если следом идет блокирующее»

В данном разделе также рассмотрена методика эвристического анализа возможных путей выполнения логических блоков. В листинге 5 приведен пример описания на языке SystemVerilog с конструкциями как нарушающими данное правило, так и соответствующими ему.

Язык SystemVerilog предоставляет три основных типа присваиваний: непрерывное (`assign`), блокирующее (=) и неблокирующее (<=), каждое из которых реализует свое поведение при выполнении операций присваивания. Из них блокирующие и неблокирующие присваивания можно использовать в процедурных блоках.

Неблокирующие присваивания выполняются одновременно в рамках процедурного блока и завершаются к концу его выполнения. Они аналогичны работе регистров в аппаратной реализации схемы. Блокирующие присваивания, напротив, работают по принципу последовательного выполнения, схожему с присваиваниями в языке программирования C.

На практике при проектировании схем блокирующие присваивания должны использоваться в комбинационной логике, а неблокирующие присваивания – в последовательностной логике (глава *Gotcha28: Blocking assignments in sequential procedural blocks* из книги [38]).

Смешение двух типов присваиваний в одном и том же блоке `always` может привести к несоответствиям в результатах симуляции исходной RTL-модели и синтезированной логической схемы. Также стоит отметить, что использование блокирующего присваивания после неблокирующего может привести к неожиданному поведению при симуляции; например, в модуле `top` из листинга 5 на строке 5 инженер-разработчик может ожидать, что в «`c`» запишется значение «`0`» после присваивания на строке 4 (начальное значение переменной «`b`»), но на самом деле это не так: в «`c`» запишется значение «`1`», так как

неблокирующие присваивания параллельно выполняются в самом конце после выполнения всех блокирующих.

```
1  module top ();
2    logic a = 1, b = 0, c = 1;
3    always_comb begin
4      a <= b;
5      c = a; // Нарушение - блокирующее присваивание после неблокирующего
6      b <= b + 1;
7    end
8
9
10   always_comb begin
11     a <= b;
12     if (b == c)
13       c = a;  // Нарушение - блокирующее присваивание после неблокирующего
14     else
15       c = 1;  // Нарушение - блокирующее присваивание после неблокирующего
16   end
17
18   always_comb begin
19     a = b;
20     c <= a; // Нарушения нет
21   end
22 endmodule
```

Листинг 5. Иллюстрация различных аспектов правила ASSIGN04.
Listing 5. Various aspects of the ASSIGN04 rule.

Правило рассматривает процедурные блоки, реализующие *комбинационную* логику, которая описывает схемы без памяти и инерции. В отличие от нее, *последовательностная* логика имеет память, синхронизирована с тактовым сигналом и зависит от предыдущего состояния, а логика *защелок* позволяет сохранять старые значения или обновлять память при определенных условиях.

Для точного разграничения типов логики в процедурных блоках были сформулированы критерии, которые учитывают используемые в них операции. Особое внимание уделено процедурным блокам `always`, которые относятся к тестовой среде. Введем несколько ключевых понятий:

Вход логики: набор символов, используемых в списке чувствительности процедурного блока;

Путь: последовательность операторов в порядке их описания при определенных значениях входов логики;

Выход логики: набор символов, значения которых изменяются внутри процедурного блока хотя бы по одному пути.

К тестовой среде относятся те блоки, которые не содержат входов (обладающий пустым списком чувствительности) или выходов, так как такой тип логики обычно является несинтезируемым большинством инструментов. Синтезируемая логика обычно имеет как входы, так и выходы. Последовательностная логика определяется наличием входа, представляющего фронт тактового сигнала или сброса, а ее выходы находятся в регистрах. Комбинационная логика характеризуется тем, что ее выходы обновляются по всем путям, но входы не содержат фронта тактового сигнала. Логика защелок определяется отсутствием фронта для входных сигналов и тем, что ее выходы обновляются лишь на части путей.

Эти критерии позволяют классифицировать и анализировать различные типы логики в контексте синтезируемых и симуляционных конструкций на языке SystemVerilog, примеры процедурных блоков, каждого типа логики представлены в листинге 6.

`always @(in, en)`
`begin`
`  if (en)`
`  out = in;`
`  else`
`  out = ~in;`
`end`
	`always @(posedge clk)`
	`begin`
	`  count <= count + 1;`
	`  r1 <= r1 - 1;`
	`  r2 <= r2 * 1;`
	`  r3 <= r3 + P;`
	`end`
		`always @(in, en)`
		`begin`
		`  if (en)`
		`  out = in;`
		`  if (en)`
		`  out1 = in;`
		`end`
			`always begin`
			`  clk = ~clk;`
			`  #5`
			`end`
			
			`always @(en)`
			`  $display(en);`

Комбинационная	**Последовательностная**	**Защелка**	**Тестовая среда**

Листинг 6. Примеры типов логики процедурных блоков.
Listing 6. Procedural logic examples.

Несмотря на то, что в стандарте языка SystemVerilog предусмотрены специальные синтаксические конструкции, обозначающие тип логики, такие как `always_comb`, `always_ff` и `always_latch`, семантические ограничения на реализацию логики не определены стандартом в полной мере. Это является лишь маркером для инструментов, работающих с описанием RTL-модели. Соответственно, реализация логики защелки в блоке `always_comb` (например, вследствие ошибки проектирования или опечатки) не является ошибочной с точки зрения стандарта. В случае отсутствия инструмента линтинга и статического анализа, который мог бы проверить такие конструкции по схожим правилам с обычными блоками `always`, недочет может быть обнаружен только на этапе логического синтеза и только в том случае, если инструмент синтеза имеет соответствующую диагностику.

В процессе проектирования детектора данного правила была разработана эвристика, определяющая достижимость путей выполнения кода в процедурном блоке. Порядок выполнения операций при вычислении пути соответствует процедурному стилю, что означает выполнение операций одна за одной в той последовательности, в которой они располагаются в исходном описании на SystemVerilog. Для решения задачи обработки путей выполнения внутри процедурного блока необходимо было решить проблему ветвления путей. Для этого был применен эвристический подход, учитывающий операторы `if`, `case` и `for`, поскольку они наиболее часто используются для ветвления в синтезируемой логике. При синтезе все условные переходы преобразуются в набор логических элементов или мультиплексоров, а циклы, обладающие конечным количеством итераций, полностью раскручиваются.

Для реализации этого детектора был создан простой менеджер ограничений, накладываемых на символы, названный `ConstraintManager`. Этот класс связывает каждый символ, используемый в выражении, с набором линейных ограничений. Линейные ограничения представлены как объединение отрезков значений, которые может принимать число. Полное множество значений, которое может принимать данный символ, определяется его типом. Например, символу `'A'` в листинге 7 соответствует линейное ограничение вида `[-32, -3]`, `[10, 31]`.

```
logic[5:0] A;
if (A <= -3 || A > 9) begin
  // -32 <= A <= -3 или 9 < A <= 31
end;
```

Листинг 7. Пример линейного ограничения внутри оператора `if`.
Listing 7. An example of a linear constraint inside an `if` statement.

`ConstraintManager` используется для определения всех возможных путей выполнения, что усложняет его использование для нескольких символов одновременно. Это происходит

потому, что линейные ограничения должны преобразовываться в декартовы произведения множеств значений.

Для определения путей выполнения кода в процедурном блоке используется обходчик областей видимости программы, представленных в АСД, и собирает информацию о видимых в ней символах и накладываемых на них новых ограничениях. Области видимости образуются модулями, функциями, задачами, блочными операторами, операторами `if`, `case` и циклов. Для модулей, функций и задач происходит сопоставление соединений, фактических аргументов – портам и формальным аргументам для передачи информации о текущих вычисленных линейных ограничениях. Операторы `if`, `case` и операторы циклов обрабатываются особым образом.

Оператор `case`: Каждая ветка оператора `case` формирует отдельную область видимости. Для каждой ветки составляется список символов, обновленных в этой ветке, включая ветку `default`. Если символ не обновлен в одной ветке, но обновлен в другой, это может указывать на логику защелки.

Оператор `if`: Накладывает линейные ограничения на символ, используемый в условии, и создает комплементарные условия для ветки `else`. Для каждой области видимости (ветки `if` и `else`) собирается информация о символах, обновленных при всех возможных значениях символа условия.

Операторы циклов: Для синтезируемости цикла он должен быть раскручиваемым. Циклы `generate for` раскручивается на этапе построения развернутой модели, то есть во время построения АСД. Раскручивание обычных циклов for является трудной задачей для инструментов логического синтеза, так как условия циклов могут быть произвольной сложности, например, использовать выражения, не зависящие от параметров цикла, и другое.

Критерии раскрутки цикла `for` включают необходимость наличия константного инициализатора для счетчика, константного шага, сравнения с константой в условии останова и отсутствие управляющих конструкций `continue` и `break`, приводящим к множественным выходам, а также отсутствие изменений счетчика в теле цикла. Если цикл не раскручиваемый, то предполагается, что символы, изменяемые внутри цикла, обновляются не по всем путям. Это связано с невозможностью гарантирования выполнения цикла конечное постоянное число раз. Для определения того, что цикл является раскручиваемым необходимо, чтобы он содержал инвариантную инициализацию и граничное условие цикла, а также имел только индуктивные переменные в выражении шага. Гнезда циклов должны раскручиваться снизу-вверх. Примеры раскручиваемых и нераскручиваемых циклов можно наблюдать в листинге 8.

```
always @(in, en) begin
  // возможно развернуть цикл
  for (int i = 0; i < 10; i ++)
    ...
  // невозможно развернуть цикл
  for (int i = in; i < 10; i ++)
    ...
  // невозможно развернуть цикл
  for (int i = 0; i < in; i ++)
    ...
end
```

Листинг 8. Пример разворачиваемых и не разворачиваемых циклов.

Listing 8. Examples of loops which can and can't be unrolled.

В SystemVerilog поддерживается множество вариантов циклических управляющих конструкций, из которых на данный момент анализируются лишь `for`, `while`, `do-while` и repeat. Циклы `forever` и `foreach` на данный момент в реализованном инструменте считаются нераскручиваемыми и отнесены к несинтезируемой логике.

Таким образом, если удается вывести ограничения на переменные по всем путям каждого выхода процедурного блока, то становится возможным определить реализуемый в блоке тип логики, и остается лишь проверить присутствие спецификации фронтов сигналов в списке чувствительности.

Для процедурных блоков, реализующих комбинационную логику, детектору нужно проверить, что на каждом пути в процедурном блоке нет блокирующего присваивания, если ранее на том же пути было встречено неблокирующее. Стоит отметить, что при рассмотрении циклов в детекторе используется эвристика, согласно которой каждое тело цикла обходится ограниченное число раз, задаваемое опцией детектора. Эта эвристика помогает отследить комбинации присваиваний, выполняемых на разных итерациях цикла. В листинге 9 представлена упрощенная реализация детектора, реализующая описанные подходы.

В этом листинге представлены обработчики двух обходчиков – `ProceduralBlockVisitor` (строки 2–9) и `AssignmentVisitor` (строки 12–51). Первый обходчик предназначен для фильтрации тех процедурных блоков, которые не реализуют комбинационную логику (строки 4–5), и для инициализации второго обходчика (строки 6–7). `AssignmentVisitor` обрабатывает основные типы управляющих конструкций в процедурных блоках комбинационной логики и выдает диагностику при нахождении блокирующего присваивания, расположенного за неблокирующим на одном пути. Путь задается как набор непротиворечивых ограничений на переменные, используемые в управляющих конструкциях. Ограничения накапливаются в структуре данных `constraintsStack`, реализованной в виде стека, которые затем проверяются на консистентность (строки 37–38) для исключения потенциально недостижимых путей. Структура данных `firstNonBlockingAssignMap` (строка 39) содержит позицию первого неблокирующего присваивания для каждого пути. Первое встреченное на пути неблокирующее присваивание помечается (строки 42–43); при обработке блокирующего присваивания проверяется, встречалось ли на текущем пути неблокирующее присваивание (строка 47), и в таком случае выдается диагностическое сообщение (строки 48–51). Обработчик на строках 12-31 предназначен для учета и распространения ограничений на диапазоны значений переменных, используемых в условных выражениях веток оператора `if`, с учетом вложенности операторов. Стоит отметить, что рассмотрение веток `else-if` было упущено в приведенном листинге с целью упрощения, рассмотрена лишь ветка `else`. Ограничения на переменные ветки `else` вычисляются как отрицание ограничений предшествующих веток оператора (строка 21). После вычисления ограничений рекурсивно вызывается обходчик `AssignmentVisitor` для каждого из путей, передавая вычисленные ограничения дальше (строки 17–18 и 23–25). После обхода всех веток оператора `if` найденная на путях информация о неблокирующих присваиваниях сохраняется в структуре данных `firstNonBlockingAssignMap`, что помогает при анализе присваиваний в управляющих операторах, обладающих зависимыми условиями на одном уровне вложенности, как в листинге 10.

Из листинга видно, что первый процедурный блок не содержит нарушений правила, так как путь, проходящий через два оператора `if`, невозможен. Второй, напротив, содержит два оператора `if`, которые, при определенных значениях `b`, могут посещаться одновременно.

В листинге 9 не представлен обходчик оператора `case`, который собирает информацию по путям каждой из его меток, включая метку по умолчанию, а также не представлен обходчик циклов. Тело каждого цикла посещается несколько раз, так как это помогает уточнить те

границы значений переменных, которые изменяются в цикле и используются вне его пределов.

Стоит отметить, что представленная выше реализация детектора является достаточно упрощенной в целях репрезентативности. Различные аспекты детектора рассматриваемого правила реализованы в инструменте существенно сложнее, например, для многобитных сигналов учитываются также присваивания значений отдельным битам сигнала.

```
// Обход по всем процедурным блокам, реализующим комбинационную логику
bool ProceduralBlockVisitor::handle(const ProceduralBlockSymbol& symbol) {
    // Фильтрация процедурных блоков
    if (!procBlocksKinds.Comb.contains(&symbol))
        return false;
    AssignmentVisitor visitor({}, {});
    symbol.getBody().visit(visitor);
    return true;
}


// Обработка условных операторов if
bool AssignmentVisitor::handle(const ConditionalStatement& cond) {
    ConstraintManager manager(context);
    Constraint ifConstraints = manager.getConstraints(cond);
    auto& newConstraintsStack = pushConstraints(constraintsStack, ifConstraints);


    AssignmentVisitor trueBranch(firstNonBlockingAssignMap, newConstraintsStack);
    cond.ifTrue.visit(trueBranch);


    // Обход else ветки
    Constraint elseConstraint = ifConstraints.getComplementConstraints();
    newConstraintsStack = pushConstraints(constraintsStack, elseConstraints);
    AssignmentVisitor falseBranch(firstNonBlockingAssignMap, newConstraintsStack);
    if (cond.ifFalse != nullptr)
        cond.ifFalse->visit(falseBranch);


    // Анализ результатов обхода и добавление новых ограничений пути
    if (!trueBranch.firstNonBlockingAssignMap.empty())
      mergeAssignMaps(firstNonBlockingAssignMap, rueBranch.firstNonBlockingAssignMap);
    return false; // Аналогично для falseBranch
}


// Выдача диагностики на операторах присваивания
bool AssignmentVisitor::handle(const AssignmentExpression& assign) {
    auto constraints = getConstraints(constraintsStack);
    // Проверка, что текущий путь достижим с точки зрения ограничений
    if (!pathConstraintCheckConsistentency(constraints))
        return false;
    auto& firstNonBlockingAssign = firstNonBlockingAssignMap[constraints];


    // Обнаружено первое неблокирующее присваивание в процедурном блоке
    if (!firstNonBlockingAssign && assign.isNonBlocking())
        firstNonBlockingAssignMap[constraints] = &assign;


    // Обнаружено блокирующее присваивание следом за неблокирующим на этом пути
    if (firstNonBlockingAssign && assign.isBlocking())
        diag(assign->location,
            "Неблокирующее присваивание не должно использоваться"
            " в комбинационной логике, если следом идет блокирующее.");
    return false;
}
```

Листинг 9. Реализация обходчиков детектора правила ASSIGN04.
Listing 9. Implementation of the ASSIGN04 rule detector visitors.

```
always @(*) begin
  a = 0;
  if (b > 5) begin
    a <= 1;
  end
  if (b < 5) begin
    a = 1; // Нарушения нет – ограничения на
           // условия путей противоположны
  end
end

always @(*) begin
  a = 0;
  if (b < 10) begin
    a <= 1;
  end
  if (b < 20) begin
    a = 1; // Нарушение – существует путь,
           // включающий оба присваивания
  end
end
```

Листинг 10. Пример нетривиальных путей в процедурных блоках.
Listing 10. An example of non-trivial paths in procedural blocks.

7. Тестирование на открытых проектах

В рамках данной работы были реализованы детекторы для 38 правил из составленного списка, описанного в разделе 4. Еще 28 покрываются уже имеющимися в slang детекторами или диагностиками, некоторые из которых требуют доработки для покрытия более сложных случаев нарушений правил. Для оценки качества реализованных детекторов использовались составленные вручную синтетические примеры как нарушающие данные правила, так и соответствующие им, а также проекты с открытым исходным кодом:

- aes [39] – реализация на Verilog симметричного блочного шифра AES – NIST FIPS 197 (Йоахим Стрембергсон);
- PicoRV32 [40] – оптимизированный по размеру процессор RISC-V (компания YosysHQ);
- black-parrot [41] – многоядерный процессор RISC-V с поддержкой Linux (компания Black Parrot);
- CVA6 [42] – 64-битный RISC-V процессор, реализующий конвейер с 6 стадиями, обладающий поддержкой Linux (компания OpenHW);
- SCR1 [43] – ядро микроконтроллера RISC-V (компания Syntacore);
- Ibex [44] – небольшое 32-битное ядро процессора RISC-V (компания lowRISC);
- VeeR EH1 [45] – 32-битное ядро RISC-V с расширениями (компания Western Digital, CHIPS Alliance);
- openC910 [46] – высокопроизводительный 64-битный многоядерный процессор, построенный на архитектуре RISC-V (компания XUANTIE-RV, Alibaba);
- ASIC bitcoin miner [47-48] – описание дизайна для майнинга биткоина (Мичиганский университет, США).

- OpenTitan [49] – платформа для создания заслуживающих доверия аппаратных компонентов (RoT, Root of Trust) (компания Google).

Тестирование осуществлялось на сервере с четырьмя 64-ядерными процессорами AMD EPYC 7662 с ограничением тактовой частоты в 2.9 ГГц под управлением операционной системы openSUSE Tumbleweed и объемом оперативной памяти 512GB.

Качество анализа оценивалось путем ручной разметки результатов анализа и подсчета точности, которая вычислялась как отношение истинных срабатываний к сумме истинных (TP, True Positive) и ложных (FP, False Positive) рассмотренных срабатываний. Детекторы с большим числом срабатываний оценивались по 100 случайным срабатываниям в различных файлах проекта. Результаты всех разработанных детекторов представлены в табл. 3.

Табл. 3. Результаты анализа разработанных детекторов системы статического анализа SVAN на проектах с открытым исходным кодом.
Table 3. SVAN static analysis system detectors open-source projects analysis results.

Проект	**Срабатываний**			**Точность**	**Время анализа, сек.**	**Количество строк, тыс.**
	Всего	Рассмотренных	FP			
aes	0	0	0	**100%**	3	3,5
PicoRV32	207	207	1	**99,5%**	4,8	7,5
black-parrot	342	174	1	**99,4%**	390	128,5
CVA6	926	536	6	**98,9%**	683	225,2
SCR1	64	64	1	**100%**	15,6	13,7
Ibex	88	84	1	**98,8%**	43,6	66,5
VeeR EH1	252	252	4	**98,4%**	137	18,3
openC910	1421	221	5	**97,7%**	4260	351,5
ASIC miner	1	1	0	**100%**	0,3	0,7
OpenTitan	670	264	5	**98,1%**	3605	587,2

Из результатов анализа следует, что разработанные детекторы демонстрируют высокую точность (около 98%) и приемлемую скорость работы (корреляция времени анализа и количества строк не является линейной, так как время анализа того или иного детектора больше зависит от сложности исходного кода схемы, чем от его размера). Основная часть ложных срабатываний связана с детекторами правил CASE01 («Каждый оператор case должен иметь метку по умолчанию») и SIGNAL07 («Не должно быть использований неинициализированного сигнала или переменной»).

В детекторе правила CASE01 был реализован подсчет перекрытия значениями меток всего диапазона значений селектора оператора `case`. В случае, когда метки полностью покрывают все значения селектора, метка по умолчанию не требуется, и, следовательно, диагностика является излишней. На момент тестирования не была учтена обработка do-not-care значений («?») в метках, что привело к некорректной диагностике ситуаций полного перекрытия в операторах `casex`/`casez`.

Детектор правила SIGNAL07 выдает ложные срабатывания при использовании переменных, которые определены как `typedef struct` и применяются в коде тестовой среды. Для этих

переменных может потребоваться нетривиальная инициализация, что не было учтено при первоначальной разработке детектора.

Все ложноположительные срабатывания были учтены и будут исправлены в рамках дальнейшей работы.

Компилятор SystemVerilog и все реализованные детекторы тестировались на разработанных вручную синтетических тестах (парсер SystemVerilog был также протестирован на автоматически сгенерированных тестах) и тестах из набора `sv-tests` [10].

Также проведено сравнительное тестирование реализованных детекторов с аналогичными детекторами встроенного линтера инструмента Verilator (см. раздел 3.2) на открытых проектах aes и PicoRV32. Следует отметить, что количество совпадающих детекторов в разработанной инфраструктуре статического анализа SVAN и Verilator невелико. Однако на совпадающих детекторах количество истинных срабатываний у SVAN выше, чем у Verilator.

В листинге 11 представлен пример нарушения правила ASSIGN03 («*Не должно быть блокирующих присваиваний в последовательностной логике*») в проекте PicoRV32 [40], которое не было обнаружено инструментом Verilator.

```
// picorv32.v
reg irq_delay;
reg irq_active;
reg [31:0] irq_mask;
reg [31:0] irq_pending;
reg [31:0] timer;
reg [31:0] next_irq_pending;
...
always @(posedge clk) begin
  ...
  if (!resetn) begin
    ...
    irq_active <= 0;
    irq_delay <= 0;
    irq_mask <= ~0;
    next_irq_pending = 0; // Нарушение правила ASSIGN03
    irq_state <= 0;
    timer <= 0;
    ...
  end
  ...
end
```

Листинг 11. Пример описания, нарушающего правило ASSIGN03 из проекта PicoRV32.
Listing 11. An example of a design violating the ASSIGN03 rule from the PicoRV32 project.

Использование блокирующих присваиваний в последовательностной логике приводит к избыточному усложнению логического синтеза из-за установления зависимостей триггеров и генерации дополнительной логики. В листинге 11 представлен процедурный блок, который был определен детектором, как реализующий последовательностную логику. Данный факт не вызывает сомнений, так как он реализует логику только на регистрах и привязан к положительному фронту тактового сигнала. В проекте PicoRV32 установлено два фрагмента описания, нарушающих правило ASSIGN03, которые не удалось обнаружить с помощью модуля линтинга инструмента Verilator, но обнаруженных детектором из инструмента SVAN. Все остальные срабатывания детекторов данного правила в этом проекте у инструментов совпадают.

Далее рассмотрены примеры нарушений других правил, найденные инструментом SVAN в проектах с открытым исходным кодом.

В листинге 12 представлен пример части описания схемы ASIC из проекта [47], который нарушает правило PORT10 (см. раздел 6.1): сигнал присоединен в обратном порядке.

В проекте black-parrot [41] также был найден интересный случай, нарушающий правило SIGNAL07 («*Не должно быть использований неинициализированного сигнала или переменной*»). Правило является достаточно актуальным, так как использование неинициализированных значений может привести к непредсказуемым результатам симуляции (моделирования). Рассмотрим пример кода в листинге 13.

```
// src/naive_design.sv
module naive_design(
    input clk,
    input reset,
    output logic done,
    output logic [255:0] H_out
);
...
endmodule


// testbench/sha_tb.sv
logic [0:255] result_test;
...
naive_design tested_module (
        .clk(clk),
        .reset(reset),
        .done(done_out_test),
        .H_out(result_test)
        // Нарушение правила PORT10:
        // подключение сигнала с обратным порядком
    );
...
```

Листинг 12. Пример описания, нарушающего правило PORT10 из проекта [47].
Listing 12. An example of a design violating the PORT10 rule from the [47] project.

```
// bp_common_rv64_pkgdef.svh
localparam dword_width_gp = 64;
// bp_be_pipe_int.sv
localparam num_bytes_lp = dword_width_gp >> 8;
...
logic [num_bytes_lp-1:0][7:0] orcb;
for (genvar i = 0; i < num_bytes_lp; i++)
  assign orcb[i] = {8{|rs1[8*i+:8]}};
  ...


always_comb
  unique case (decode.fu_op)
    // Нарушение правила SIGNAL07
    e_int_op_orcb : alu_result = orcb;
```

Листинг 13. Пример описания, нарушающего правило SIGNAL07 из проекта black-parrot.
Listing 13. An example of a design violating the SIGNAL07 rule from the black-parrot project.

В листинге 13 переменная `orcb` типа `logic` инициализируется побитово в цикле непосредственно после объявления, но, если заметить, что количество итераций цикла зависит от переменной `num_bytes_lp`, значение которой равно `64` деленное на `256` или `0`, что эквивалентно битовому сдвигу вправо на `8`. Это значит, что цикл, зависимый от генерируемой переменной `i`, будет развернут нулевое количество раз. Следовательно, ни один бит переменной `orcb` инициализирован не будет.

Данная ошибка была найдена разработчиками независимо от авторов статьи и была исправлена в недавнем изменении [50].

Также в этом проекте было найдено нарушение правила CASE03 («*Ширина выражения выбора в case не соответствует ширине вариантов выбора*»). Несовпадение битовой ширины селектора и перечисления может привести к отклонению от ожидаемых результатов симуляции или синтеза. Рассмотрим его подробнее в листинге 14.

```
// bp_common_rv64_pkgdef.svh
module bp_fe_icache
  always_comb
    for (integer i = 0; i < assoc_p; i++)
      case (tag_mem_pkt_cast_i.opcode)
        e_cache_tag_mem_set_tag:
          ...
        e_cache_tag_mem_set_state:
          ...
        {1'b0, e_cache_tag_mem_set_inval}: // Нарушение правила CASE03
          ...
      endcase


// bp_common_cache_engine_pkgdef.svh
typedef enum logic [2:0]
{
  e_cache_tag_mem_set_clear, e_cache_tag_mem_set_tag,
  e_cache_tag_mem_set_state, e_cache_tag_mem_set_inval,
  e_cache_tag_mem_read
} bp_cache_tag_mem_opcode_e;
```

Листинг 14. Пример описания, нарушающего правило CASE03 из проекта black-parrot.
Listing 14. An example of a design violating the CASE03 rule from the black-parrot project.

В листинге 14 выражение селектора оператора `case` является полем `opcode` переменной `tag_mem_pkt_cast_i` и имеет тип `bp_cache_tag_mem_opcode_e`, который соответствует перечисляемому типу с трехбитными значениями. Все, кроме одной, метки являются элементами данного перечисления. В последней метке используется выражение – конкатенация элемента перечисления и однобитного нуля. Соответственно, ширина последней метки равна четырем, что не соответствует ширине выражения селектора, которое равное трем.

О данном нарушении было сообщено разработчикам [51], а также было опубликовано его исправление [52].

При анализе проекта CVA6 в одном из используемых им компонентов, а именно в компоненте, реализующем контроллер прерываний `rv_plic` [53], было найдено нарушение правила LATCH01 («*Логику защелки не следует использовать в описании, кроме тех случаев, где это действительно необходимо*»). Появление непреднамеренной защелки обычно вызвано ошибками проектирования, например, когда отсутствует значение по умолчанию для переменной, переопределяемой в условном операторе (за исключением случаев, когда это явно подразумевается). Рекомендуется избегать появления защелок (кроме тех случаев, где это действительно необходимо) в описании схем, так как они не синхронизированы с тактовыми сигналами, что влечет к излишнему усложнению схемы. Рассмотрим листинг 15.

В листинге 15 приведен пример описания из автоматически генерируемого файла, в котором инженером-проектировщиком явно используется процедурный блок для описания комбинационной логики `always_comb`, но один из выходов, а именно однобитный `ip_re_o`, не получает начальное значение в процедурном блоке, но существует условное присваивание в этот выход по одной из меток оператора `case`. Для данного примера инструмент

логического синтеза сгенерирует защелку, то есть элемент последовательностной логики, что, вероятно, не соответствует намерению проектировщика.

Данная ошибка была найдена разработчиками независимо от авторов статьи и была исправлена в изменении [54].

```
// plic_regmap.sv
module plic_regs (
  output logic [30:0] prio_re_o,
  output logic [0:0] ip_re_o,
  output logic [1:0][30:0] ie_o)
  // предполагалась комбинационная логика
  always_comb begin
    prio_re_o = '0;
    ie_o = '0;
    ... // нет предварительной инициализации ip_re_o
    unique case(req_i.addr)
      ...
      32'hc000078: begin
        resp_o.rdata[2:0] = prio_i[30][2:0];
        prio_re_o[30] = 1'b1;
      end
      32'hc001000: begin
        resp_o.rdata[30:0] = ip_i[0][30:0];
        ip_re_o[0] = 1'b1; // Нарушение правила LATCH01
      end
...
```

Листинг 15. Пример описания, нарушающего правило LATCH01 из проекта CVA6.
Listing 15. An example of a design violating the LATCH01 rule from the CVA6 project.

8. Заключение

В данной работе представлена разработанная система статистического анализа SVAN, которая состоит из компилятора SystemVerilog, предназначенного для построения развернутого представления цифровых описаний аппаратуры на языках Verilog и SystemVerilog, и модуля анализа для проверки структурных и семантических свойств описаний. Также на основании опыта инженеров российских дизайн-центров электроники, был составлен список проблем, возникающих в ходе проектирования и разработки цифровых схем. Для каждой проблемы было разработано соответствующее правило. Эти правила затем были классифицированы по 18 категориям. Для 38 правил были разработаны детекторы с помощью инфраструктуры статического анализа SVAN, которые затем были протестированы на синтетических тестах и проектах описаний дизайнов с открытым исходным кодом. На синтетических примерах разработанные детекторы не имели ложных срабатываний, а на представленных проектах с открытым исходным кодом точность анализа составила порядка 98%. По сравнению с инструментом Verilator разработанные детекторы дают более точные и полные результаты.

В рамках дальнейшей работы планируется продолжить реализацию детекторов для разработанных правил, а также реализовать инфраструктуру для разработки детекторов, требующих анализ потока данных и символьное выполнение, с использованием инструментов CIRCT и KLEE.

Также планируется провести сравнительное тестирование разработанных детекторов с коммерческими анализаторами, обладающими детекторами аналогичных правил.

Список литературы / References

[1]. Riesgo T., Torroja Y., de la Torre E. Design Methodologies Based on Hardware Description Languages. IEEE Transactions on Industrial Electronics, vol. 46, no. 1, 1999. pp. 3-12. DOI: 10.1109/41.744370.

[2]. Earlham College. – Intel Pentium fdiv error, https://www.cs.earlham.edu/~dusko/cs63/fdiv.html, accessed 01.12.2024

[3]. Mcilroy R., Sevcik J., Tebbi T., Titzer B.L., Verwaest T. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv:1902.05178, 2019. https://arxiv.org/abs/1902.05178.

[4]. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017. 1315 p. DOI: 10.1109/IEEESTD.2018.8299595.

[5]. IEEE Standard for VHDL Language Reference Manual. IEEE Std 1076-2019. 673 p. DOI: 10.1109/IEEESTD.2019.8938196.

[6]. Смолов С.А. Обзор методов извлечения моделей из HDL-описаний. Труды ИСП РАН, том 27, вып. 1, 2015. с. 97-124. DOI: 10.15514/ISPRAS-2015-27(1)-6.

[7]. "IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[8]. EBMC – model checker for hardware designs, accessed 12.04.2024. https://www.cprover.org/ebmc, accessed 01.12.2024

[9]. Zhang R., Deutschbein C., Huang P., Sturton C. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018. pp. 815-827. DOI: 10.1109/MICRO.2018.00071.

[10]. sv-tests – SystemVerilog parsing synthetic test suit. https://github.com/chipsalliance/sv-tests, accessed 01.12.2024

[11]. SpyGlass: Early Design Analysis Tools for SoCs. https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html, accessed 01.12.2024

[12]. VCS: Functional Verification Solution. https://www.synopsys.com/verification/simulation/vcs.html, accessed 01.12.2024

[13]. IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE Std 1800.2-2020. 458 p. DOI: 10.1109/IEEESTD.2020.9195920.

[14]. Jasper Formal Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, accessed 01.12.2024

[15]. Xcelium logic simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, accessed 01.12.2024

[16]. Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/, accessed 01.12.2024

[17]. Icarus Verilog, https://steveicarus.github.io/iverilog/, accessed 01.12.2024

[18]. Verible – SystemVerilog parser and linter, https://github.com/chipsalliance/verible, accessed 01.12.2024

[19]. svlint – SystemVerilog linter, https://github.com/dalance/svlint, accessed 01.12.2024

[20]. SystemVerilog parser library, https://github.com/dalance/sv-parser, accessed 01.12.2024

[21]. Verilator – SystemVerilog simulation tool, https://github.com/verilator/verilator, accessed 01.12.2024

[22]. KLEE – symbolic virtual machine built on top of the LLVM compiler infrastructure, https://github.com/klee, accessed 01.12.2024

[23]. Surelog – SystemVerilog 2017 pre-processor, parser, elaborator, UHDM compiler, https://github.com/chipsalliance/Surelog, accessed 01.12.2024

[24]. Dargelas A., Zeller H. Universal Hardware Data Model. Workshop on Open-Source EDA Technology (WOSET), 2020. https://woset-workshop.github.io/PDFs/2020/a10.pdf

[25]. CIRCT – Circuit IR Compilers and Tools, https://github.com/llvm/circt, accessed 01.12.2024

[26]. Lattner C. et al. MLIR: A compiler infrastructure for the end of Moore's law // arXiv preprint arXiv:2002.11054. – 2020.

[27]. Mike Popoloski, slang - SystemVerilog Language Services, https://github.com/MikePopoloski/slang, accessed 01.12.2024

[28]. Mike Popoloski, slang-tidy - SystemVerilog linter, https://github.com/MikePopoloski/slang/tree/master/tools/tidy, accessed 01.12.2024

[29]. slang based frontend for Yosys, https://github.com/povik/yosys-slang, accessed 01.12.2024

[30]. Takemoto T. Joint activity for semiconductor R&D and role of Semiconductor Technology Academic Research Center (STARC). International Symposium on VLSI Technology, Systems, and Applications, 2001. pp. 7-10. DOI: 10.1109/VTSA.2001.934468.
[31]. Synopsis Inc., Mentor Graphics Corporation (2001). OpenMORE Assessment Program for Hard/Soft IP Version 1.0, http://www.openmore.com, accessed 01.12.2024
[32]. Hilderman V., Baghi T. Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware). Avionics Communications, 2007. 244 p.
[33]. Clang: a C language family frontend for LLVM, https://clang.llvm.org, accessed 01.12.2024
[34]. LLVM Language Reference Manual, https://llvm.org/docs/LangRef.html, accessed 01.12.2024
[35]. Moore Dialect, https://circt.llvm.org/docs/Dialects/Moore/, accessed 01.12.2024
[36]. Erhart M., Schuiki F., Yedidia Z., Healy B., Grosser T. Arcilator: Fast and cycle-accurate hardware simulation in CIRCT. LLVM Developers Meeting, 2023. https://llvm.org/devmtg/2023-10/slides/techtalks/Erhart-Arcilator-FastAndCycleAccurateHardwareSimulationInCIRCT.pdf
[37]. The RTL Intermediate Language (RTLIL), https://yosyshq.readthedocs.io/projects/yosys/en/stable/yosys_internals/formats/rtlil_rep.html, accessed 01.12.2024
[38]. Sutherland S., Mills D. Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them. Springer, 2010. 218 p.
[39]. AES – Hardware implementation of AES encryption algorithm, https://github.com/secworks/aes, accessed 01.12.2024
[40]. PicoRV32 – A Size-Optimized RISC-V CPU, https://github.com/YosysHQ/picorv32, accessed 01.12.2024
[41]. Black Parrot – A Linux-Capable Accelerator Host RISC-V Multicore, https://github.com/black-parrot/black-parrot, accessed 01.12.2024
[42]. The CORE-V CVA6 is an Application class 6-stage RISC-V CPU capable of booting Linux, https://github.com/openhwgroup/cva6, accessed 01.12.2024
[43]. SCR1 – RISC-V Core, accessed 10.06.2024. https://github.com/syntacore/scr1, accessed 01.12.2024
[44]. Ibex – RISC V Core, https://github.com/lowRISC/ibex, accessed 01.12.2024
[45]. VeeR EH1 Core, https://github.com/chipsalliance/Cores-VeeR-EH1, accessed 01.12.2024
[46]. OpenXuantie – OpenC910 Core, https://github.com/XUANTIE-RV/openc910, accessed 01.12.2024
[47]. ASIC Design for Bitcoin Mining, https://github.com/susansun1999/eecs570_final_project, accessed 01.12.2024
[48]. Sun Y., Yang H., Zhang W., Gu Y. ASIC Design for Bitcoin Mining. University of Michigan, 2021. https://zwtaoumich.github.io/paper/EECS570_Final_Report.pdf
[49]. OpenTitan: Open source silicon root of trust (RoT). https://opentitan.org/, accessed 01.12.2024
[50]. black-parrot, Hotfix: orbc typo, https://github.com/black-parrot/black-parrot/pull/1228/commits/87866ddfa37ab9d5df8e5951d3b7865b23676c77, accessed 01.12.2024
[51]. Case label selector mismatch. https://github.com/black-parrot/black-parrot/issues/1230, accessed 01.12.2024
[52]. Fix case label mismatch. https://github.com/black-parrot/black-parrot/pull/1232, accessed 01.12.2024
[53]. RISC-V Platform-Level Interrupt Controller. https://github.com/pulp-platform/rv_plic/, accessed 01.12.2024
[54]. Adjust codegen to fix synthesis latch. https://github.com/pulp-platform/rv_plic/pull/7, accessed 01.12.2024

Информация об авторах / Information about authors

Ян Андреевич ЧУРКИН – младший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации, языки описания аппаратуры.

Yan Andreevich CHURKIN – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations, hardware description languages.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Константин Николаевич КИТАЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Konstantin Nikolaevich KITAEV – student at MIPT, laboratory assistant at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Алексей Георгиевич ВОЛОХОВ – ведущий инженер отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Aleksey Georgievich VOLOKHOV – leading engineer at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Егор Викторович ДОЛГОДВОРОВ – студент МФТИ, Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Egor Viktorovich DOLGODVOROV – student at MIPT. Research interests: static analysis, compiler technologies, optimizations.

Александр Сергеевич КАМКИН – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий программирования ИСП РАН, ведущий научный сотрудник РЭУ им. Г.В. Плеханова. Научные интересы: формальные методы, синтез и верификация цифровой аппаратуры, гетерогенные компьютерные системы.

Alexander Sergeevich KAMKIN – Cand. Sci. (Phys.-Math.), leading researcher at Software Engineering department of ISP RAS, leading researcher at Plekhanov Russian University of Economics. Research interests: formal methods, synthesis and verification of digital hardware, and heterogeneous computer systems.

Артем Михайлович КОЦЫНЯК – научный сотрудник отдела технологий программирования ИСП РАН. Научные интересы: формальные методы, компиляторные технологии, модели вычислений, языки программирования.

Artem Mikhailovich KOTSYNYAK – researcher at Software Engineering department of ISP RAS. Research interests: formal methods, compiler technologies, models of computation, programming languages.

Дмитрий Олегович САМОВАРОВ – стажер-исследователь отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации, языки описания аппаратуры.

Dmitry Olegovich SAMOVAROV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations, hardware description languages.

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

CC BY

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.

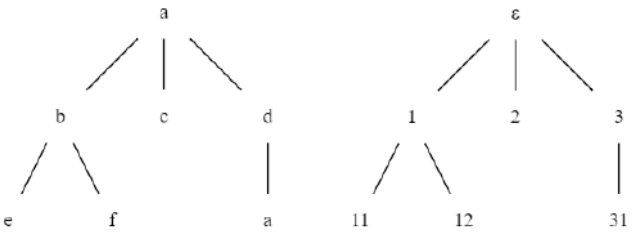


Рис. 1. Дерево $a(b(ef)cd(a))$ и соответствующие позиции вершин.
Fig. 1. The $a(b(ef)cd(a))$ tree and positions of corresponding vertices.

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. □

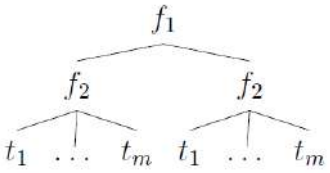


Рис. 4. Дерево $f_1(f_2(t_1 \dots t_m)f_2(t_1 \dots t_m))$.
Fig. 4. The $f_1(f_2(t_1 \dots t_m)f_2(t_1 \dots t_m))$ tree.

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.

DOI: 10.15514/ISPRAS-2025-37(1)-3

CC BY

Компьютерное моделирование и оптимизация конструкции микрофлюидного чипа

А.А. Варфоломеева, ORCID: 0009-0004-8931-5646 <aavarfolomeeva@edu.hse.ru>
Л.А. Пятко, ORCID: 0009-0002-6321-2732 <lapyatko@edu.hse.ru>
С.Р. Паршина, ORCID: 0009-0008-2011-4220 <srparshina@edu.hse.ru>

Национальный исследовательский университет "Высшая Школа Экономики",
Россия, 123458, г. Москва, ул. Таллинская, д. 34.

Аннотация. Работа посвящена анализу гидродинамической модели клапана Теслы, позволяющего регулировать поток жидкости в конструкции микрофлюидного чипа. При моделировании движения жидкости в сложных клапанах Теслы, состоящих из нескольких петель, часто прибегают к анализу одной петли и дальнейшей экстраполяции результатов на весь клапан. Для подтверждения или опровержения корректности данного метода, в работе анализируются клапаны Теслы состоящие из одной и восьми петель. Для них была изучена возникающая разность давлений и диодичность. На основе полученных данных было выявлено, что гидродинамика клапана Теслы является нелинейной и не позволяет корректно обобщать результаты, полученные для одной петли, на клапаны, состоящие из большего числа петель.

Ключевые слова: микрофлюидика; гидродинамика; клапан Теслы; расчетная платформа Ansys Workbench; платформа численного моделирования OpenFOAM.

Для цитирования: Варфоломеева А.А., Пятко Л.А., Паршина С.Р. Компьютерное моделирование и оптимизация конструкции микрофлюидного чипа. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 55–64. DOI: 10.15514/ISPRAS–2025–37(1)–3.

Modeling and Analysis of the Microfluidic Chip

A.A. Varfolomeeva, ORCID: 0009-0004-8931-5646 <aavarfolomeeva@edu.hse.ru>
L.A. Pyatko, ORCID: 0009-0002-6321-2732 <lapyatko@edu.hse.ru>
S.R. Parshina, ORCID: 0009-0008-2011-4220 <srparshina@edu.hse.ru>
National Research University Higher School of Economics,
Tallinskaya Street, 34, Moscow, 123458, Russia.

Abstract. The work is devoted to the analysis of the hydrodynamic model of the Tesla valve, which allows to regulate the flow of liquid in the design of a microfluidic chip. When modelling fluid motion in complex Tesla valves consisting of several loops, one often resorts to analyzing a single loop and further extrapolating the results to the entire valve. To confirm or refute the validity of this method, valves consisting of one and eight loops are analyzed in the work. The resulting pressure difference and diodicity were studied. Based on the obtained data, it was revealed that the hydrodynamics of the Tesla valve is nonlinear and does not correctly generalize the results obtained for one loop to valves consisting of a larger number of loops.

Keywords: microfluidics; hydrodynamics; Tesla valve; Ansys Workbench; OpenFOAM.

For citation: Varfolomeeva A.A., Pyatko L. A., Parshina S.R. Modelling and analysis of the microfluidic chip. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1 , 2025, pp. 55-64 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-3.

1. Введение

Микрофлюидика – быстро развивающаяся наука, которая стала неотъемлемой частью многих инженерных и биомедицинских приложений [1]. Ее применяют повсеместно, например, в технологии «орган на чипе» (OOAC) – новой разработке, основанной на микрофлюидных платформах и культивировании клеток in vitro [2]. Разработаны устройства – «отек легких на чипе», изготовленное с использованием методов мягкой литографии [3], «легочные дыхательные пути на чипе» для исследования взаимодействия между гладкомышечными и эпителиальными клетками [4] и «сердце на чипе» для получения сердечной ткани из новорожденных крыс и кардиомиоцитов [5]. В 2001 году было предложено экспериментальное устройство для наблюдения за трубчатым потоком почечных клеток [6], а позднее предложили конструкцию «почки на чипе», основанную на микроэлектромеханической системе (МЭМС) [7]. Микрожидкостная культура клеток позволяет контролировать поток жидкости в микрометровом и нанолитровом масштабе и облегчает анализ экспериментов, начиная с уровня отдельной клетки и заканчивая крупными популяциями клеток и тканями [8].

Микрофлюидный чип представляет собой систему каналов и полостей, позволяющую реализовывать различные операции выделения, сортировки и анализа биологической пробы [4, 8, 9]. Микроканалы формируются в пластине полидиметилсилоксана (PDMS), которая наклеивается на предметное стекло. Для обеспечения движения жидкости внутри каналов используется система пневмонасосов, мембраны которых также формируются в PDMS пластине.

Клапан Теслы в макромасштабе был изобретен Николой Теслой в 1920 году для применения в газовых турбинах [10]. В связи с растущим интересом к микроэлектромеханическим системам (MEMS) [11, 12] в 1990-х годах этот тип клапанов был применен для микрожидкостных систем [13]. В результате множества исследований было выяснено, что клапан Теслы без движущихся частей гораздо более устойчив к износу и усталости, а расход для насоса (объем жидкости, перекачиваемый оборудованием в единицу времени) с такой формой почти в три раза больше, чем у аналогичного насоса, но с клапанами неоптимизированной формы [14-16]. Проводились исследования по оптимизации геометрии клапана (рис. 1) путем изменения внутреннего радиуса, угла и прямого сегмента

устройства [17-19].

В работе [20] была предложена трехмерная параметрическая модель клапана Теслы (рис. 2), которая показывает лучшие результаты работы с точки зрения диодичности (характеристики отношения обратного потока к прямому) относительно предыдущей геометрии (рис. 1). Для этой модели проводились исследования по оптимизации параметров, а именно подбирались оптимальное количество петель клапана [21, 22] и длина прямого сегмента устройства [23]. Исследования проводились с целью изучения клапана Теслы, напоминающего по форме жидкостный выпрямитель [24, 25].

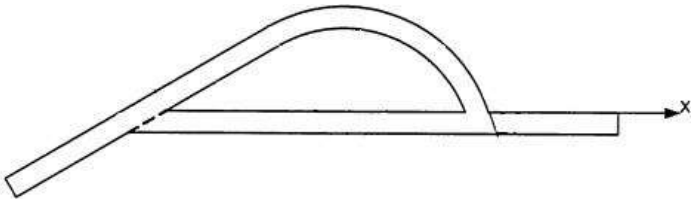


Рис. 1. Геометрия клапана Теслы [17].
Fig. 1. The geometry of the Tesla valve [17].

Позже конструкция клапана на рис. 2 использовалась при проверке эффективности оптимизированной абляции в режиме фемтосекундной записи, для чего использовался клапан Теслы, состоящий из 8 петель (рис. 4) [26]. В работах [17, 20] вместо моделирования всего клапана Теслы, состоящего из 8 петель, исследуется одна петля, после чего полученные данные экстраполируются на клапан целиком.

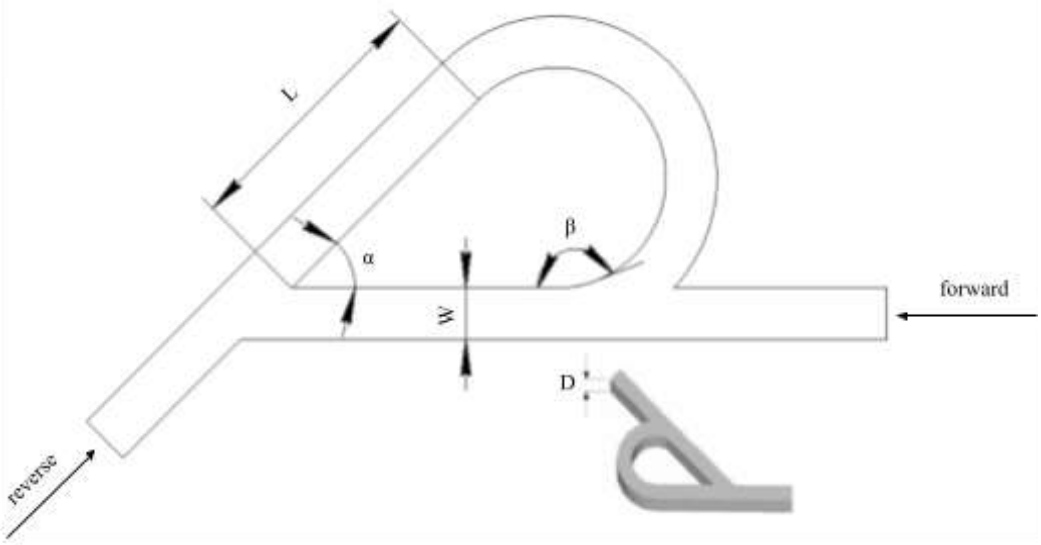


Рис. 2. Схематическое изображения клапана Теслы.
Fig. 2. Schematic representation of the Tesla valve.

2. Подготовка конечно-элементной модели

Конечно-элементная модель была построена с использованием расчетной платформы Ansys Workbench Fluent методом MultiZone с опцией Hexa. Метод MultiZone основан на блочной технологии и обеспечивает автоматическое разделение геометрии на две группы геометрических тел: объёмы, к которым может быть применен метод Sweep для построения гексасетки с протягиванием сеточных элементов вдоль некоторой оси, и остальные объемы,

для которых строится гексасетка неструктурированного типа.

Для геометрии, представленной на рис. 2, были построены 3 сетки методом MultiZone с опцией Неха с заданной длиной ребра каждого элемента 6, 4.5 и 3 мкм, чтобы определить, с какого момента сетка становится достаточно точной для моделирования (рис. 3). Экспериментальным путем было выяснено, что, начиная с длины ребра 4.5 мкм, разница между значениями, полученными в результате моделирования, отличается меньше, чем на 2%. Было решено использовать длину грани 4.5 мкм для построения сеток.

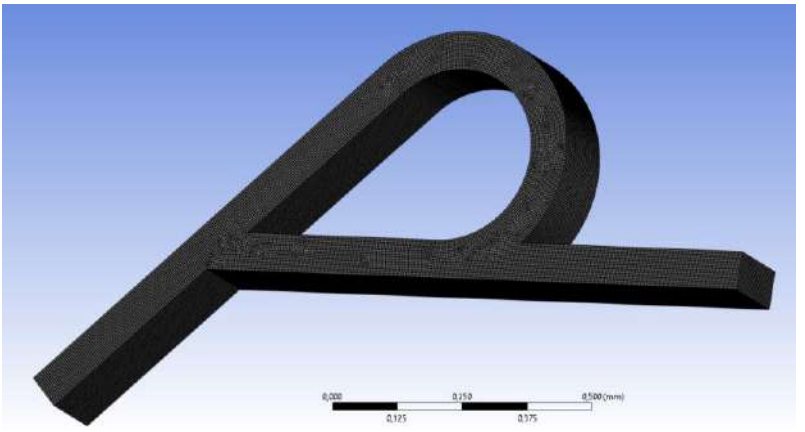


Рис. 3. Конечно-элементная модель клапана Теслы.
Fig. 3. Finite element model of the Tesla valve.

Для модели клапана с резервуарами (рис. 4) использовалось несколько разных длин: на клапане Теслы длина ребра каждого элемента составляла 6 мкм, на резервуарах – 400 мкм и на клапане длина грани 12 мкм, а на резервуарах – 400 мкм. Также после проведения экспериментов была достигнута точность в 10% и далее использовалась сетка с длинами ребер 6 мкм на клапане и 400 мкм на цилиндрах.

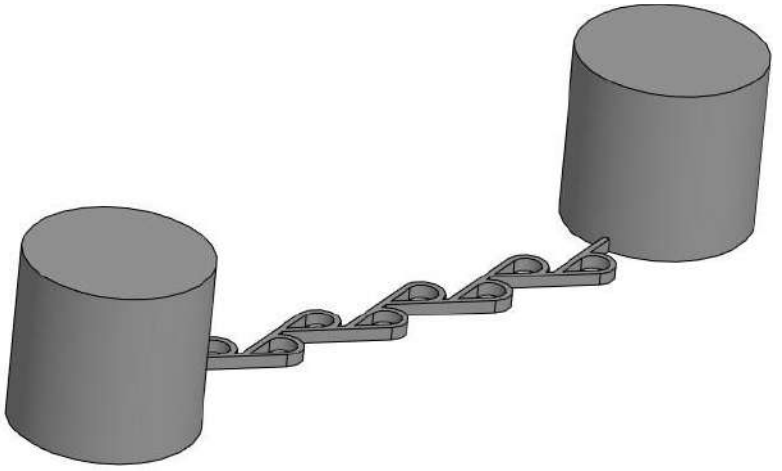


Рис. 4. Модель клапана Теслы из 8 петель.
Fig. 4. Model with receptacles.

Для решения гидродинамической задачи течения жидкости в трехмерном клапане используется свободно распространяемый пакет с открытым исходным кодом OpenFOAM.

Для решений уравнений Навье-Стокса выбран алгоритм SIMPLE, предназначенный для задач со стационарными течениями несжимаемой изотропной жидкости.

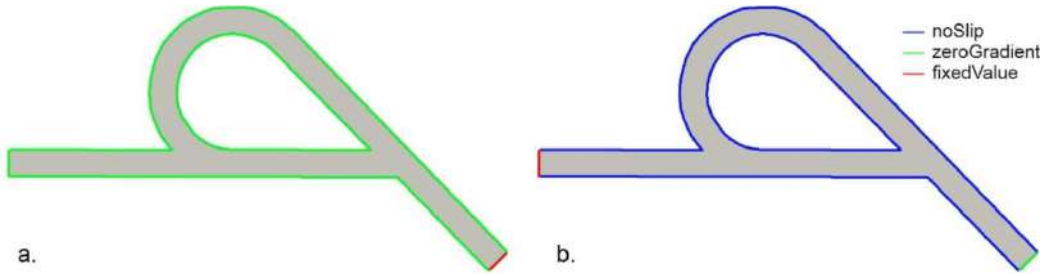


Рис. 5. Граничные условия для a) давления, b) скорости.
Fig. 5. Boundary conditions for a) pressure, b) velocity.

В качестве сплошной среды использовалась вода при температуре 20ºC (с плотностью ρ = 998.2 кг/м³ и динамической вязкостью $\mu = 1.003\times10^{-3}$ кг/м·с). Геометрическая модель одной петли клапана Теслы соответствует приведенному чертежу на рис. 2 с параметрами: D = 150 мкм, W = 75 мкм, L = 400 мкм, $\alpha = 45°$, $\beta = 20°$. При моделировании использовалась трехмерная модель клапана, и устанавливались следующие граничные условия в случае прямого потока: для входа и боковых стенок применялись граничные условия первого и второго рода, а для выхода – граничные условия второго и первого рода для скорости (рис. 5a) и давления (рис. 5b) соответственно. При этом давление на выходе считалось равным 0 Па, а скорость на входе была равномерно распределена по всей грани и находилась в диапазоне от 1 мл/мин до 3.5 мл/мин. При такой скорости число Рейнольдса варьировалось между 148 и 519. При изменении направления потока с прямого на обратный осуществляется инверсия граничных условий для скорости и давления, заданных на входной и выходной границах.

3. Результаты

3.1 Анализ клапана Теслы с одной и восемью петлями

При изучении влияния клапанов Теслы на характеристики системы, часто является целесообразным редуцировать систему до более простых фрагментов, моделируя и исследуя которые, можно делать выводы о поведении системы целиком. Одним из возможных вариантов является моделирование одной петли клапана Теслы. Тогда, согласно гипотезе, разность давлений P создаваемая n одинаковыми петлями можно описать следующей формулой: $P = n \cdot P_0$, где P_0 – разность давлений, создаваемая одной петлей. Подобный метод позволяет существенно уменьшить компьютерное время, затрачиваемое методом контрольных объемов, но в то же время, ввиду общей нелинейности процессов гидродинамики, остается не до конца понятным, насколько редукция системы влияет на погрешность итоговых результатов. Для проверки справедливости гипотезы, описанной выше, были исследованы две модели клапана Теслы: с одной и восемью петлями.

По результатам моделирования можно сказать, что распределение скорости в петлях клапана Теслы с 8 петлями (рис. 6a) практически совпадает с распределением скорости в модели с одним клапаном Теслы (рис. 6b). Также можно заметить, что каждая петля (рис. 6c и 6d) создает разность давлений между участками канала до и после него. Однако, как оказалось, зависимость между количеством петель и возникающей разностью давлений нельзя назвать линейной. Так, например, для скорости 3.5 мл/мин разность давлений, полученная с помощью модели с восемью петлями на 6.5% больше, чем разность давлений, вычисленная из модели с одной (рис. 7). Эта же величина на 40.2% меньше в случае прямого потока. Можно сделать вывод, что линейная модель не позволяет делать правильные выводы о процессах, протекающих в клапанах Теслы, состоящих из нескольких петель.

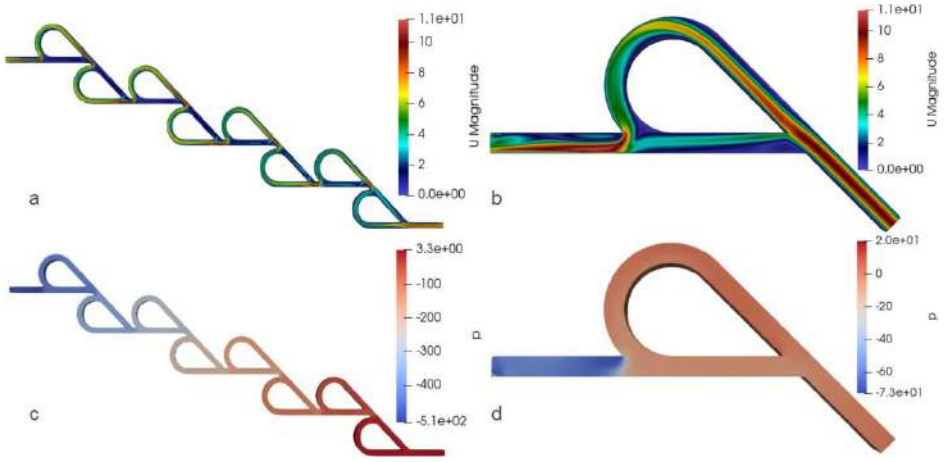


Рис. 6. Результаты моделирования для v=3.5 мл/мин в обратном потоке скорость, 8 петель, b) скорость, одна петля, c) кинематическое давление, 8 петель, d) кинематическое давление, одна петля.
Fig. 6. Simulation results for v=3.5 ml/min in reverse flow a) velocity, 8 loops b) velocity, 1 loop, c) kinematic pressure, 8 loops, d) kinematic pressure, 1 loop.

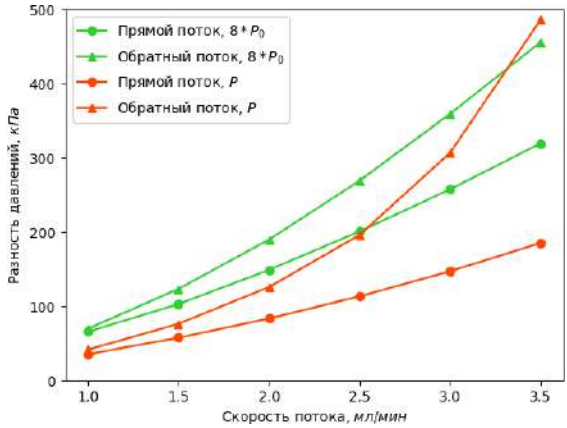


Рис. 7. Разность давлений для двух моделей.
Fig. 7. Pressure difference for two models.

Также, для каждой из моделей можно вычислить её диодичность. Эта величина является отношением разности давлений обратного и прямого потока, и часто служит как показатель эффективности клапана Теслы:

$$D(v) = \frac{\Delta P^{-}(v)}{\Delta P^{+}(v)}$$

Из рис. 8 видно, что диодичность одного клапана лишь немного превосходит 1, в то время как каскад из 8 клапанов увеличивает диодичность до 2.83, то есть весьма значительно.

3.2 Анализ клапана Теслы с добавленными резервуарами по его краям и без них

Для валидации модели с реальными данными было проведено сравнение результатов моделирования с экспериментальной работой [26]. В ней жидкость попадала в канал не

напрямую, а через резервуары с обеих сторон канала. Для корректного сравнения была построена модель, учитывающая особенности проведения эксперимента (рис. 4). Для проведения оценки был выполнен расчет на крайних значениях скорости, поскольку с усложнением модели возрастает вычислительная нагрузка. Визуализация результатов моделирования для геометрии с резервуарами в обратном потоке при расходе 3.5 мл/мин представлены на рисунке 9a для распределения поля скоростей и на рисунке 9b для распределения кинематического давления.

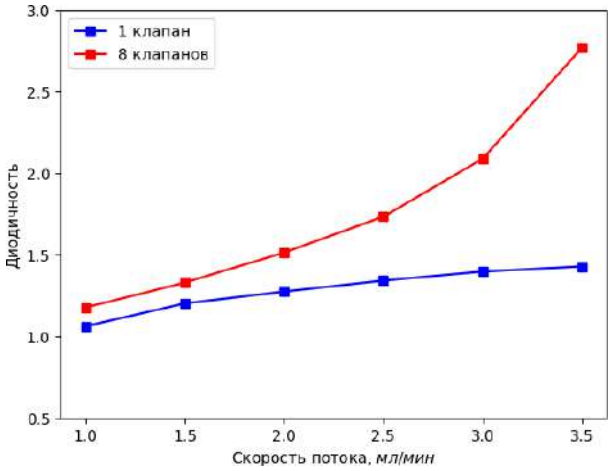


Рис. 8. Диодичность для канала с 1 и 8 клапанами Теслы.
Fig. 8. Diodicity for channel with 1 and 8 Tesla valves.

Результаты показали, что диодичность в случае отсутствия резервуаров отличается на 51.3%, а с резервуарами – на 30.2% от экспериментальных данных (табл. 1). Средние значения перепада давления отличаются на 67% (табл. 2) как для геометрии без резервуаров, так и для геометрии с ними в сравнении с результатами эксперимента. Такие значения ошибок не позволяют подтвердить валидность построенной модели. Сравнительный анализ между геометриями с резервуарами и без показывает, что разница диодичности при минимальной скорости составляет 0.86%, а при максимальной – 23.6%. При скорости 1 мл/мин обе модели демонстрируют схожие результаты, при этом разница в возникающих давлениях не превышает 2%. Однако при скорости 3.5 мл/мин различия становятся более выраженными. В частности, разница давления в обратном потоке в модели без резервуаров на 16.6% превышает таковую в модели с резервуарами, тогда как в прямом потоке наблюдается противоположная ситуация: разница давления в модели без резервуаров на 11% меньше, чем в модели с ними.

4. Заключение

В результате исследования было выяснено, что моделирование одной петли клапана Теслы не позволяет делать корректные выводы о физико-механических свойствах всей исследуемой области. При использовании предположения, что разность давлений линейно зависит от количества петель клапана, погрешность измерений может достигать 40%. Также было изучено, насколько условие проведения эксперимента – попадание жидкости не напрямую в канал, а через резервуары – может повлиять на измерение разности давлений и диодичности. Оказалось, что для небольшого расхода (1 мл/мин) разность давлений в моделях с резервуарами и без них отличается не более чем на 2%. Однако при скорости 3.5 мл/мин различия в разности давлений в обеих моделях может доходить до 17%, а диодичность на 31% больше в модели без резервуаров, чем с ними.

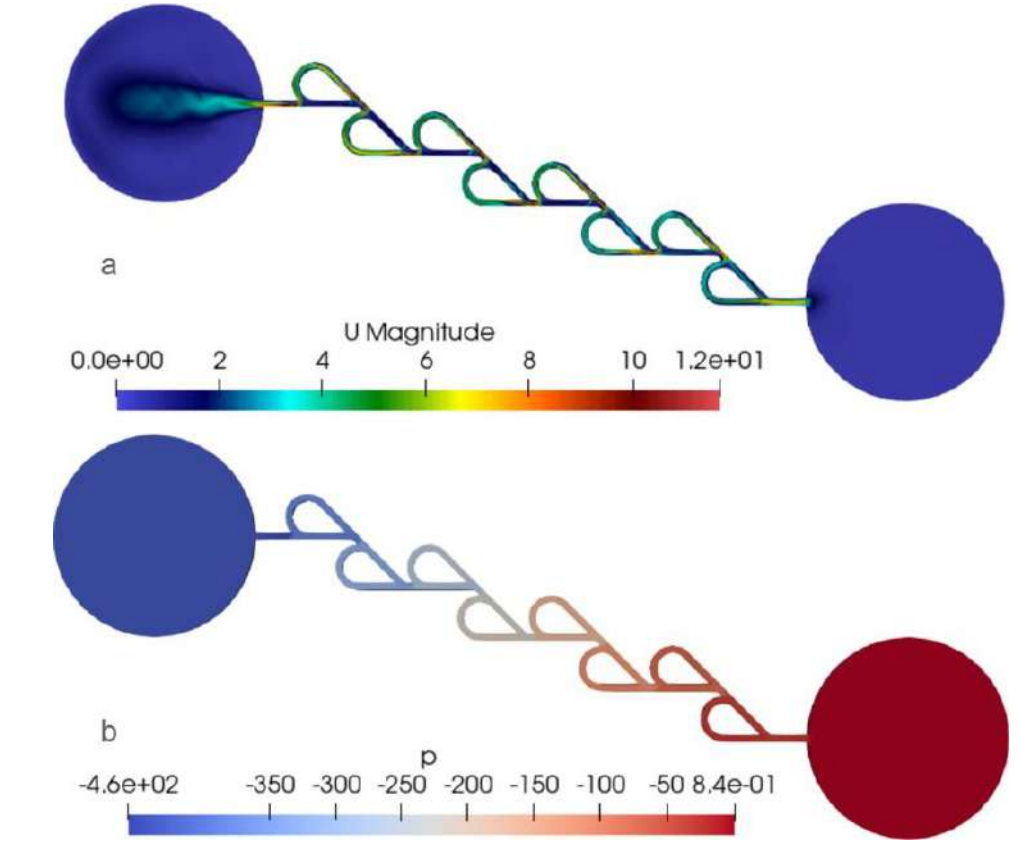


Рис. 9. Результаты для модели с резервуарами в обратном потоке, v = 3.5 мл/мин
a) скорость, b) кинематическое давление.
Fig. 9. Simulation results for a model with tanks in reverse flow, v = 3.5 ml/min
a) velocity, b) kinematic pressure.

Табл. 1. Диодичность. Сравнение результатов с экспериментом [26].
Table 1. Diodicity. Comparison of the results with the experiment [26].

	Di		
v, мл/мин	8 клапанов	8 клапанов + резервуары	статья
1	1.16	1.15	0.90
3.5	2.83	2.16	1.63
Разница со статьей [26]	29.1%	28.0%	
	73.5%	32.5%	

Табл. 2. Перепад давления. Сравнение результатов с экспериментом [26].
Table 2. Pressure drop. Comparison of the results with the experiment [26].

	p, кПа					
	8 клапанов		8 клапанов + резервуары		статья	
v, мл/мин	forward	reverse	forward	reverse	forward	reverse
1	28.8	33.4	29.3	33.7	100	90
3.5	148.2	418.9	166.5	359.3	620	1010
Сравнение результатов со статьей [26]	71.2%	62.9%	70.7%	62.5%		
	76.1%	58.5%	73.1%	64.4%		

Список литературы / References

[1]. Nguyen, N.-T.; Wereley, S.T. Fundamentals and Applications of Microfluidics. Artech House, 2002.
[2]. Mufeeda C. Koyilot, Priyadarshini Natarajan, Clayton R. Hunt, Sonish Sivarajkumar,1 Romy Roy, Shreeram Joglekar, Shruti Pandita, Carl W. Tong, Shamsudheen Marakkar, Lakshminarayanan Subramanian, Shalini S. Yadav, Anoop V. Cherian, Tej K. Pandita, Khader Shameer, Kamlesh K. Yadav Breakthroughs and Applications of Organ-on-a-Chip Technology. Cells, vol. 11, 2022. 1828 p.
[3]. Huh, D.; Matthews, B.D.; Mammoto, A.; Montoya-Zavala, M.; Hsin, H.Y.; Ingber, D.E. Reconstituting organ-level lung functions on a chip. Science, vol. 328, pp. 1662-1668.
[4]. Humayun, M.; Chow, C.W.; Young, E.W.K. Microfluidic lung airway-on-a-chip with arrayable suspended gels for studying epithelial and smooth muscle cell interactions. Lab Chip, vol. 18, 2018, pp. 1298-1309.
[5]. Marsano, A.; Conficconi, C.; Lemme, M.; Occhetta, P.; Gaudiello, E.; Votta, E.; Cerino, G.; Redaelli, A.; Rasponi, M. Beating heart on a chip: A novel microfluidic platform to generate functional 3D cardiac microtissues. Lab Chip., vol. 16, 2016, pp. 599-610.
[6]. Essig, M.; Terzi, F.; Burtin, M.; Friedlander, G. Mechanical strains induced by tubular flow affect the phenotype of proximal tubular cells. Am. J. Physiol. Renal Physiol, vol. 281, 2001, pp. 751-762.
[7]. Weinberg, E.; Kaazempur-Mofrad, M.; Borenstein, J. Concept and computational design for a bioartificial nephron-on-a-chip. Int. J. Artif. Organs, vol. 31, 2008, pp. 508-514.
[8]. Matthias Mehling, Savas¸ Tay Mehling, M. Microfluidic cell culture. Current Opinion in Biotechnology, vol. 25, 2014, pp. 95-102.
[9]. Tatara, A.M. Role of Tissue Engineering in COVID-19 and Future Viral Outbreaks. Tissue Eng., vol. A(26), 2020, pp. 468-474.
[10]. Tesla N. Valvular conduit. U.S. Patent No. 1329559, 1920.
[11]. Mcdonald, J.C.; Duffy, D.C.; Anderson, J.R.; Chiu, D.T.; Wu, H.; Schueller, O.J.; Whitesides, G.M. Fabrication of microfluidic system in poly (dimethylsiloxane). Electrophoresis., vol. 21, 2015, pp. 27-40.
[12]. Weng, X.; Yan, S.; Zhang, Y.; Liu, J.; Shen, J. Design, simulation and experimental study of a micromixer based on Tesla valve structure. Chem. Ind. Eng. Prog., vol. 40, 2021, pp. 4173-4178.
[13]. Forster F.K., Bardell R.L., Afromowitz M.A. and Sharma N.R. Design, fabrication and testing of fixed-valve micro-pumps. Proc. ASME Fluids Engineering Division, ASME International Mechanical Engineering Congress and Exposition (San Francisco), vol. 234, 1995, pp. 39-44.
[14]. Gamboa, A.R.; Morris, C.J.; Forster, F.K. Improvements in fixed valve micropump performance through shape optimization of valves. J. Fluids Eng., vol. 127, 2005, 339 p.
[15]. Agnes Purwidyantri, Briliant Adhi Prabowo Tesla Valve Microfluidics: The Rise of Forgotten Technology. Chemosensors, vol. 11, 2023, 256 p.
[16]. Raymond H. W. Lam, Wen J. Li A Digitally Controllable Polymer-Based Microfluidic Mixing Module Array. Micromachines, vol. 3, 2012, pp. 279-294.
[17]. T-Q. Truong, N-T. Nguyen Simulation and optimization of Tesla Valves. Nanotech, vol. 1, 2003, 1 p.

[18]. Fred K. Forster, Ronald L. Bardell, Martin A. Afromowitz, Nigel R. Sharma, Alan Blanchard Design, fabrication and testing of fixed-valve micro-pumps. ASME Fluids Engineering Division, vol. 234, 1995, pp. 39-44.
[19]. Zhi-jiang Jin, Zhi-xin Gao, Min-rui Chen, Jin-yuan Qian Parametric study on Tesla valve with reverse flow for hydrogen decompression. Elsevier International Journal of Hydrogen Energy, vol. 43, 2018, pp. 8888-8896.
[20]. S. Zhang, S. H. Winoto and H.T. Low Performance Simulations of Tesla Microfluidic Valves. MicroNanoChina, vol. 21107, 2007, pp. 15-19.
[21]. S. M. Thompson, B. J. Paudel, T. Jamal, D. K. Walters Numerical Investigation of Multistaged Tesla Valves. Journal of Fluids Engineering, vol. 136, 2014, 9 p.
[22]. Benoit Scheid On the diodicity enhancement of multistage Tesla valves. Physics of Fluids, vol. 35, 2023, 5 p.
[23]. Faras Al Balushi, Arash Dahi Taleghani A Reversible Miniaturized Tesla Valve. ASME Open J. Engineering, vol. 3, 2024, 10p.
[24]. Kazem Mohammadzadeh, Amin Kolahdouz, Ebrahim Shirani, Mohammad Behshad Shafii Numerical Investigation on The Effect of The Size and Number of Stages on The Tesla Microvalve Efficiency. Journal of Mechanics, vol. 1, 2013, pp. 1-8.
[25]. Reed, J. L. and Fla, O., Fluidic Rectifier. U. S. Patent., 1993.
[26]. Deividas Andriukaitis, Rokas Vargalis, Lukas Šerpytis, Tomas Drevinskas, Olga Kornyšova, Mantas Stankevičius, Kristina Bimbiraitė-Survilienė, Vilma Kaškonienė, Audrius Sigitas Maruškas, Linas Jonušauskas Fabrication of Microfluidic Tesla Valve Employing Femtosecond Bursts. Micromachines, vol. 13(8), 2022, pp. 1180.

Информация об авторах / Information about authors

Анастасия Андреевна ВАРФОЛОМЕЕВА – студентка НИУ ВШЭ, обучается на программе бакалавриата по направлению «Прикладная математика». Сфера научных интересов: Вычислительная гидродинамика и микрофлюидные устройства.

Anastasia Andreevna VARFOLOMEEVA is enrolled in the bachelor's degree program in Applied Mathematics. Research interests: Computational fluid dynamics and microfluidic devices.

Лев Андреевич ПЯТКО – студент НИУ ВШЭ, обучается на программе бакалавриата по направлению «Прикладная математика». Сфера научных интересов: Вычислительная гидродинамика и микрофлюидные устройства.

Lev Andreevich PYATKO is enrolled in the bachelor's degree program in Applied Mathematics. Research interests: Computational fluid dynamics and microfluidic devices.

Софья Романовна ПАРШИНА – студентка НИУ ВШЭ, обучается на программе бакалавриата по направлению «Прикладная математика». Сфера научных интересов: Вычислительная гидродинамика и микрофлюидные устройства.

Sofya Romanovna PARSHINA is enrolled in the bachelor's degree program in Applied Mathematics. Research interests: Computational fluid dynamics and microfluidic devices.

DOI: 10.15514/ISPRAS-2025-37(1)-4

CC BY

Применение формальных спецификаций системы команд для функционального тестирования языковых виртуальных машин

А.С. Проценко, ORCID: 0009-0001-4240-2986 <protsenko@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В настоящее время большой популярностью пользуются языки программирования, использующие в своей инфраструктуре языковые виртуальные машины (ВМ), что стимулирует развитие данной технологии. Разработка языковой ВМ – сложный процесс, в ходе которого могут вноситься ошибки в проектируемую систему. Для обеспечения качества реализации ВМ процесс разработки включает в себя этап тестирования. При тестировании необходимо ответить на два основных вопроса: как создавать тестовые программы и как проверять результат их исполнения. В статье представлен метод функционального тестирования языковых ВМ на основе формальных спецификаций системы команд. В работе описана реализация предложенного подхода. На основе документации ВМ специфицируется системы команд с помощью языка описания архитектуры. На основе спецификации системы команд строится исполнимая модель. Для автоматизации создания тестовых программ используются тестовые шаблоны – параметризованные описания тестовых программ. При создании тестовых шаблонов используется специальный предметно-ориентированный язык, позволяющий задавать различные техники генерации и использовать байт-код ВМ, полученный из формальных спецификаций. В представленном методе тестовые шаблоны могут быть описаны вручную, сгенерированы автоматически, в соответствии с целевым критерием, или получены от сторонних генераторов. На основе тестовых шаблонов и исполнимой модели генерируются тестовые программы на байт-коде, нацеленные на проверку определенной функциональности или свойств тестируемой системы. Байт-код является естественным языком для ВМ и позволяет воздействовать на всю ее функциональность. Тестовая программа транслируется в бинарную программу и исполняется на ВМ. Во время исполнения программы на ВМ собирается трасса исполнения. Для анализа трассы исполнения создается адаптер трасс. На основе исполнимой модели и адаптера трасс строится тестовый оракул. Оракул проверяет результаты тестирования путем сравнения трассы исполнения с результатами исполнения бинарной программы на исполнимой модели. Метод реализован в инструменте MicroTESK версии 2.6 и был использован для тестирования ВМ Ark.

Ключевые слова: тестирование на основе модели; языковая виртуальная машина; ВМ; система команд; архитектуры системы команд ISA; байт-код; формальные спецификации; тестирование; тестовая программа; тестовый оракул.

Для цитирования: Проценко А.С. Применение формальных спецификаций системы команд для функционального тестирования языковых виртуальных машин. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 65–86. DOI: 10.15514/ISPRAS–2025–37(1)–4.

Functional Testing of Language Virtual Machines Based on Formal ISA Specifications

A.S. Protsenko ORCID: *0009-0001-4240-2986 <protsenko@ispras.ru>*

Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Nowadays, programming languages that use language virtual machines (VMs) in their infrastructure are very popular, which stimulates the development of this technology. Developing a language VM is a complex process, during which errors can be introduced into the designed system. To ensure the quality of the VM implementation, the development process includes a testing stage. During testing, it is necessary to answer two main questions: how to create test programs and how to check the result of their execution. The article presents a method for functional testing of language VMs based on formal specifications of the instruction set architecture (ISA). The work describes the implementation of the proposed approach. Based on the VM documentation, the ISA is specified using the architecture description language. An executable model is built based on the ISA specification. Test templates, which are parameterized descriptions of test programs, are used to automate the creation of test programs. When creating test templates, a special domain-specific language is used, which allows you to specify various generation techniques and use the VM bytecode obtained from formal specifications. In the presented method, test templates can be described manually, generated automatically in accordance with the target criterion, or obtained from third-party generators. Based on the test templates and the executable model, test programs are generated in bytecode aimed at checking a certain functionality or properties of the system under test. Bytecode is a natural language for the VM and allows you to affect all of its functionality. The test program is translated into a binary program and executed on the VM. During the program execution, an execution trace is collected on the VM. A trace adapter is created to analyze the execution trace. A test oracle is built based on the executable model and the trace adapter. The oracle checks the test results by comparing the execution trace with the results of executing the binary program on the executable model. The method is implemented in the MicroTESK tool version 2.6 and was used to test the Ark (Panda) VM.

Keywords: model-based testing; language virtual machine; VM; ISA; bytecode; formal specification; testing; test oracle.

For citation: Protsenko A.S. Functional testing of language virtual machines based on formal ISA specifications. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 65-86 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-4.

1. Введение

Концепция виртуальной машины (ВМ) была предложена около 50 лет назад и продолжает быть популярной и активно используемой. В настоящее время существует несколько вариантов классификации ВМ [1-2]. При этом выделяют два основных вида ВМ: системные ВМ (system virtual machine) и процессовые ВМ (process virtual machine). Системные ВМ (VirtualBox, QEMU, Xen и др.) предназначены для виртуализации всей компьютерной системы. Процессовые виртуальные машины предназначены для запуска отдельного приложения. Среди процессовых ВМ можно выделить большую группу языковых ВМ (Dalvik VM, Python VM, Java VM и др.), которые могут быть частью инфраструктуры ВМ для высокоуровневых языков программирования (High-Level Language VM).

В работе рассматриваются языковые ВМ, использующие систему команд, называемую байт-кодом (Java bytecode, Python bytecode, ARK Bytecode и др.). Концепция архитектуры системы команд (ISA) была описана впервые в работе [3] для системы команд микропроцессоров. Систему команд ВМ иногда называют виртуальной системой команд (virtual ISA, V-ISA).

Разработка языковой виртуальной машины – сложный процесс, в ходе которого могут вносится ошибки в проектируемую систему. Для обеспечения качества реализации ВМ в процесс разработки можно включить функциональное тестирование. Такое тестирование

обычно осуществляют при помощи: исполнения тестовых программ на тестируемой системе и проверки полученных результатов. Основными проблемами, рассматриваемыми в работе, являются: создание тестовых программ и проверка результата их исполнения на целевом устройстве.

Тестовая программа представляет собой тестовые воздействия, нацеленные на проверку определенной функциональности и/или аспекта тестируемой системы. Для автоматизации создания тестовых программ могут использоваться подходы на основе тестовых шаблонов. Тестовый шаблон представляет собой параметризованное описание структуры и функциональности тестовой программы. При этом тестовые шаблоны для ВМ должны иметь возможность использования метаданных, которые описывают классы, поля, методы, интерфейсы, объекты и другие артефакты программы. При тестировании ВМ можно выделить следующие области и уровни тестирования: транслятор, интерпретатор, загрузчик классов, верификатор байт-кода, сборщик мусора, модель памяти, оптимизатор, Just-in-Time (JIT) и Ahead-of-Time (AOT) компиляторы.

Байт-код для ВМ обычно получается с помощью трансляции с языка высокого уровня (Java, Kotlin, ArkTS). Для удачных реализация ВМ могут быть создано несколько языков высокого уровня, имеющих свои особенности при трансляции в байт-код. Поэтому для доступа ко всей функциональности ВМ необходимо использовать естественный для нее язык, а именно байт-код ВМ.

Источником информации о байт-коде и архитектурных особенностях ВМ является документация архитектуры системы команд. Поэтому методы, использующие архитектуру системы команд, могут быть реконфигурируемы, то есть настроены на систему команд, с помощью ее спецификаций.

Для проверки результата тестирования ВМ нужен тестовый оракул. Впервые термин “тестовый оракул” (test oracle) был использован в 1978 году [4]. Механизм работы оракула для ВМ заключается в сравнении результата исполнения тестовой программы на тестируемой системе и эталонного результата, который может быть получен с помощью исполнения программы на эталонной системе. Получение эталонной системы является непростой задачей, для решения которой могут быть использованы формальные спецификации.

В данной статье представлен метод функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд. Метод заключается в следующем.

Шаг 1. На основе документации ВМ специфицируем систему команд ВМ. Для этого необходимо спроектировать и реализовать хранилище метаданных для классов, полей классов, методов, обработчиков исключений, объектов и полей объектов; необходимо специфицировать алгоритмы работы с метаданными: изменение стека фреймов при вызове и возврате из метода, создание объекта и поля объекта, вызов и возврат из метода, работа с полями объекта и статическими полями класса, загрузка метаданных в память для классов, полей классов, методов, обработчиков исключений, генерация и выброс исключения и др.

Шаг 2. На основе формальных спецификаций получаем исполнимую модель (симулятор).

Шаг 3. Описываем или генерируем тестовые шаблоны. Для описания тестовых шаблонов используется специализированный предметно-ориентированный язык (DSL) способный использовать техники генерации, языковые конструкции системы команд ВМ и метаданные. Тестовые шаблоны можно разрабатывать вручную, создавать с использованием сторонних генераторов или автоматически генерировать на основе анализа спецификаций в соответствии с заданным критерием тестового покрытия.

Шаг 4. На основе тестовых шаблонов и исполнимой модели генерируем тестовые программы.

Шаг 5. Исполняем полученные тестовые программы на ВМ. Получаем бинарный образ программы (*.class, *.abc и пр.) и трассу исполнения.

Шаг 6. Для анализа трассы исполнения и извлечения событий в трассе описывается адаптер трасс.

Шаг 7. На основе исполнимой модели и адаптера трасса создается оракул.

Шаг 8. Используем оракул для проверки результатов выполнения тестовых программ на ВМ.

Метод не привязан к конкретной ВМ, применим для большинства указанных составляющих ВМ и позволяет создать инструментарий для тестирования ВМ на всех этапах ее разработки и эксплуатации. Схема метода приведена на рис. 1.

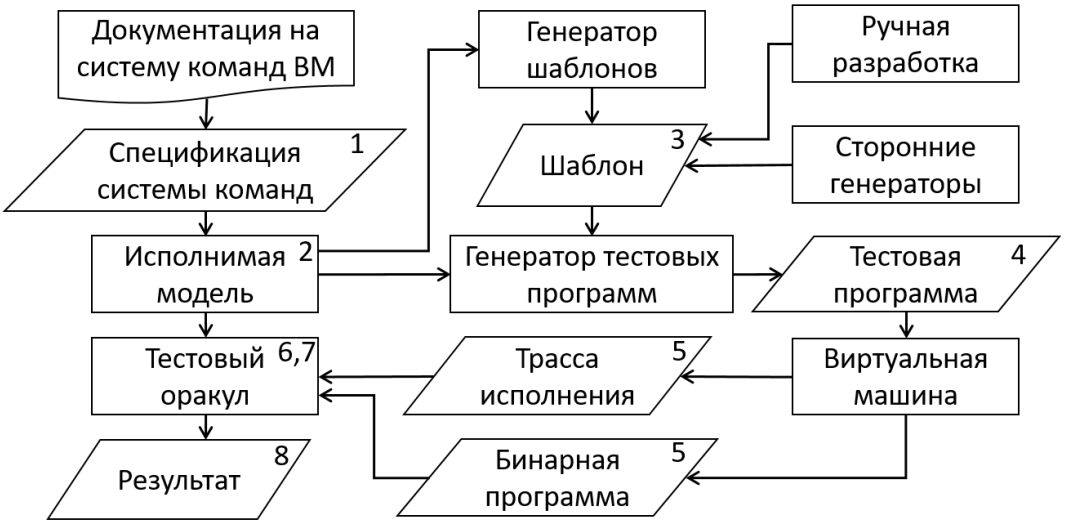


Рис. 1. Общая схема предлагаемого метода.
Fig. 1. General scheme of the proposed method.

Вклад статьи заключается в следующем:

1. Предложен метод функционального тестирования языковых ВМ на основе формальных спецификаций системы команд, впервые применяемый к тестированию ВМ.
2. Предложен метод спецификации систему команд ВМ на языке описания архитектуры.
3. Предложены методы создания тестовых программ на основе шаблонов и спецификации системы команд ВМ.
4. Предложен метод автоматизированного построения оракула на основе спецификации системы команд ВМ, для проверки результатов исполнения тестовых программ.
5. Описанные методы реализованы и применены к верификации системы команд ВМ Ark [5], где были найдены несоответствия в спецификации системы команд.

Остальная часть статьи организована следующим образом. В разделе 2 приводится обзор литературы по теме статьи. В разделе 3 описаны предлагаемые методы и их работа на примере ВМ Ark. В разделе 4 описаны полученные результаты. В разделе 5 приводится заключение.

2. Обзор литературы

Обзор литературы проводился в открытых источниках по работам, в которых описывались методы тестирование ВМ и методы похожие на предлагаемый в данной работе.

2.1 Похожая область, другие методы

В статье [6] описан метод тестирования, основанный на конколическом анализе путей исполнения инструкций в интерпретаторе и подготовке тестов, основанных на этой информации, для тестирования JIT-компилятора. Для проверки результатов сравниваются исполнения тестов интерпретатором в режимах с включенным и отключенным JIT-компилятором.

В статье [7] авторы предлагают подход тестирования Java Virtual Machine (JVM) [8], в основе которого лежат порождающие грамматики. В работе описывается предметно-ориентированный язык lava, применяемый для описания порождающих грамматик, с помощью которых генерируются тестовые программы.

В работах [9-14] описываются подходы для фаззинг тестирования ВМ. В качестве оракула используются ВМ разных версий и от разных компаний, реализующих одну спецификацию. Исходные программы для фаззинга в проектах берутся из различных открытых источников или генерируются на основе грамматических описаний.

В статье [15] проверяется корректность работы JIT оптимизации, путем проверки состояния стека ВМ при динамической JIT деоптимизации в случае нарушения охранных условий при выполнении оптимизированного кода.

В работе [16] проверяются ошибки, связанные с использованием некорректных типов операндов байт-код инструкций JVM. Авторы разделяют кодировки байт-код инструкций по группам в соответствии с типами операндов. Далее они комбинируют инструкции между собой и составляя пары. Полученные комбинации проверяют методом проверки моделей на формальной модели JVM описанной на NuSMV и получают информацию о корректности комбинаций инструкций. С помощью BCEL создают программы на байт-коде и проверяют их с помощью встроенного верификатора. Результат верификатора сравнивают с результатами от NuSMV, которые должны совпадать.

В диссертации [17] представлен метод тестирования ВМ, полученной с помощью среды генерации ВМ на основе моделирования. В качестве основы тестового набора используются тесты, созданные для проверки модели ВМ в среде моделирования. Этот набор расширяется тестами, полученными на их основе с помощью мутаций. Для проверки результата исполнения тестовых программ используется подход на основе дифференциального сравнения, в котором сравниваются результаты, полученные от сгенерированной ВМ на языке С и от ВМ из среды моделирования.

2.2 Другая область, похожие методы

Генераторы тестовых программ успешно используются для тестирования микропроцессоров. Различные подходы для построения тестовых программ были реализованы в инструментах: RIS (Random Instruction Sequence) от компании ARM [18], RAVEN (Random Architecture Verification Machine) от Obsidian Software (позже ARM) [19], Genesys-Pro используемый в IBM [20], прототип генератора MA2TG [21], генератор MicroTESK до версии 2.4 от ИСП РАН [22-25]. В данных работах для генерации тестовых программ обычно используются тестовые шаблоны, в которых есть возможность применения различных техник генерации и задания инструкций, которые необходимо добавить в тестовую программу. Некоторые инструменты могут быть настроены на целевую систему команд путем ее спецификации или с помощью описания семантики. Для проверки результатов тестирования обычно используют сравнение трасс, полученных от тестируемой RTL-модели и эталонной реализации (например, на языке С).

Основным отличием предлагаемого в данной работе подхода является: автоматическое построение оракула для проверки трасс исполнения тестовых программ, наличие механизмов работы с метаданными, автоматическое построение тестовых шаблонов. Оракул, в одном из

режимов работы, может проверять трассы с пропусками, что позволяет проверять трассы с использованием JIT и AOT компиляторов.

2.3 Похожие общие части

В работах [26-27] описано применение SMT-солверов для генерации данных для тестовых случаев. Для этого формальные предусловия переводятся в набор ограничений, разрешаемых SMT-солвером. Результаты выполнения таких тестов анализируются с помощью оракулов, основанных на формальных описаниях постусловий.

3. Описание метода и пример реализации

В разделе описаны детали предлагаемого метода функционального тестирования языковых ВМ и приведены примеры их реализации. В разделе представлены методы спецификации системы команд ВМ (раздел 3.1), создания тестовых программ на основе шаблонов и спецификации системы команд ВМ (раздел 3.2), автоматизированного построения оракула на основе спецификации системы команд ВМ (раздел 3.3).

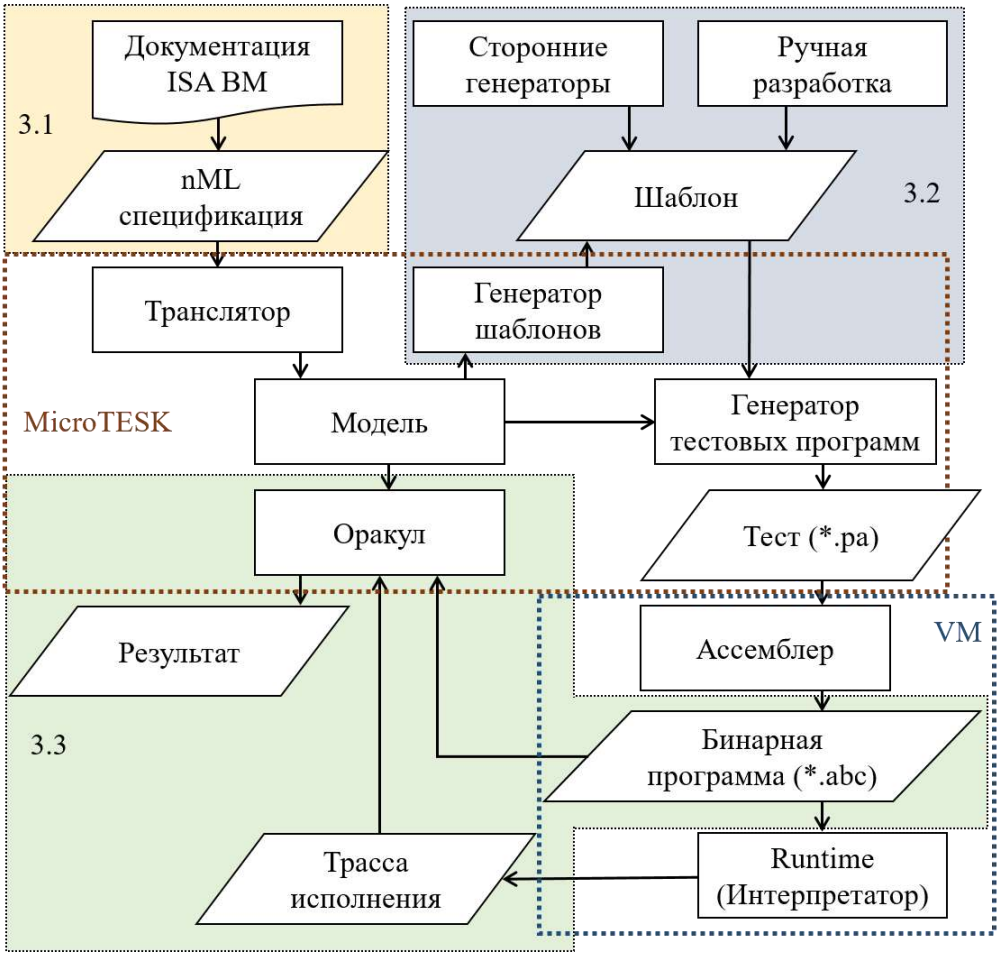


Рис. 2. Схема реализации метода функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд.
Fig. 2. Scheme of implementation of the method for functional testing of language virtual machines based on formal ISA specifications.

На рис. 2 представлена расширенная схема основного метода с учетом деталей реализации. Метод реализован на основе инструмента с открытым исходным кодом MicroTESK [28]. Ранее инструмент MicroTESK использовался для тестирования микропроцессоров. Для возможности его применения для тестирования ВМ были внесены изменения, которые реализованы в версии 2.6. Описанный метод был применен к тестированию ВМ Ark [5], фрагменты реализации некоторых элементов из этого проекта будут использоваться в качестве примеров.

3.1 Спецификация системы команд на языке nML

3.1.1 Описание модели ВМ

Модель ВМ можно описать с помощью состояния и инструкций, изменяющих это состояние.

Состояние представляет собой память ВМ и включает в себя:

- глобальную статическую память, которая содержит метаданные, используемые для описания классов, методов, полей классов, обработки исключений и пр.;
- глобальную динамическую память, которая хранит данные артефактов, порождаемых во время работы ВМ и счетчик команд PC;
- локальную память, которая организована в виде стека, элементами которого являются фреймы, которые создаются при вызове метода ВМ и содержит данные необходимые для выполнения метода: регистры, стек операндов, стек локальных переменных и пр. После возвращения из метода фрейм уничтожается.

Инструкции описывают изменение состояния ВМ. Инструкции можно разбить на две группы:

- локальные инструкции, не зависящие от метаданных и не влияющие на глобальную память. Такие инструкции чаще всего представлены арифметическими инструкциями, инструкциями ветвления и инструкциями присваивания;
- глобальные инструкции, использующие метаданные и взаимодействующие с глобальной памятью, как статической, так и динамической. Такие инструкции создают экземпляры объектов, вызывают методы и осуществляют запись в поля объектов.

Инструкции производят операции над входными данными. Входные данные передаются в инструкцию через параметры. Параметры инструкции могут быть:

- явными (прямыми) – являющимися частью сигнатуры, в этом случае эти параметры являются и частью кодировки инструкции;
- не явными (косвенными) – использующие для передачи данных память, специальные регистры или стек. Такие параметры не являются частью кодировки инструкции.

Частью исполнения некоторых инструкций является генерация и выброс исключений. Если инструкция выбрасывает исключение, то она становится источником исключения. Исключение представляет собой экземпляр специального класса, содержащего описания исключительной ситуации в методе или инструкции. Для выброса исключения в коде метода используется специальная инструкция (обычно имеющая имя throw). Такая инструкция принимает созданный ранее экземпляр класса исключения как параметр. При выбросе исключения во время исполнения инструкции, например, инструкции деления (div), экземпляр класса исключения создается автоматически перед выбросом исключения.

Для обработки исключения используются try/catch блоки. Описание try/catch блоков содержит идентификатор метода, которому принадлежит блок обработки исключений, диапазон адресов метода, в котором перехватывается исключение, идентификатор класса

исключения для обработки и адрес обработчика исключения. После выброса исключения в блоке try в диапазоне указанных адресов, проверяется его соответствие типу перехватываемого исключения, при соответствии типа, управление передается по адресу обработчика исключения. Для одного диапазона адресов может быть описано несколько try/catch блоков. Если ни один из существующих try/catch блоков для данного диапазона адресов не смог обработать исключение, то метод завершает работу, а инструкция, вызвавшая данный метод, становится новым источником созданного исключения. Экземпляр исключения будет обработан try/catch блоками метода или программа завершится с ошибкой, описание которой будет взято из экземпляра исключения.

3.1.2 Метод спецификации системы команд ВМ

В качестве входной информации используется документация с описанием системы команд и архитектуры целевой ВМ.

Метод спецификации системы команд ВМ состоит из следующих шагов:

Шаг 1. На основе документации ВМ проектируется и специфицируется структура хранилища метаданных для классов, предков классов, методов классов, параметров методов классов, полей классов, объектов, полей объектов, обработчиков исключений. Для различных ВМ структура хранилища метаданных может различаться.

Шаг 2. Реализуются примитивы, необходимые для спецификации операций по работе с метаданными: генерация идентификаторов классов, методов, полей классов, объектов, полей объектов, обработчиков исключений; выделение памяти для метаданных классов, методов, полей классов, объектов, полей объектов, обработчиков исключений; завершение работы программы.

Шаг 3. Специфицируется основные операции для добавления, модификации и чтения метаданных: изменение стека фреймов при вызове и возврате из метода; создание объекта и поля объекта; вызов метода и возврат из метода; работа с полями объекта и статическими полями класса; загрузка метаданных в память для классов, полей классов, методов, обработчиков исключений; генерация и выброс исключения. Для спецификации операций используются определенные ранее примитивы и хранилище метаданных.

Шаг 4. Специфицируется локальные инструкции системы команд ВМ.

Шаг 5. Специфицируется глобальные инструкции системы команд ВМ. Используются операции для работы с метаданными, определенные ранее.

3.1.3 Пример nML спецификаций

Для спецификации состояния и инструкций ВМ был использован диалект языка описания архитектуры nML [29-31] разработанный в ИСП РАН. Ниже приведены примеры nML кода для различных элементов спецификации регистровой ВМ Ark.

Объявление общих регистров:

```
reg R [RF_SIZE, i64]
```

Объявление масок для регистров:

```
reg R_MASK [RF_SIZE, u64]
```

Объявление хранилища для типов хранимых данных.

```
var R_TYPE[RF_SIZE, u64]
```

Маски регистров показывают значащие биты, хранящиеся в регистре. Это необходимо для корректной работы оракула, так как регистры могут содержать значения разного типа, при этом часть битов регистров может быть не определена, и при сравнении оракулам эти значения в трассе и модели могут расходится. Константа `RF_SIZE` содержит значение

количества создаваемых регистров и масок, типы `i64` и `u64` соответствуют знаковому и беззнаковому 64 битному числу.

Для доступа к регистрам используются режимы доступа, их спецификация на nML представлена ниже:

```
mode V4 (i: card(4)) = R[i + REGISTER_SIZE*FC]
  init   = { register_mode = V4_MODE; }
  syntax = format("v%d", i)
  image  = format("%4s", i)
  action = { r_index = coerce(u16, i); }

mode V8 (i: card(8)) = R[i + REGISTER_SIZE*FC]
  init   = { register_mode = V8_MODE; }
  syntax = format("v%d", i)
  image  = format("%8s", i)
  action = { r_index = coerce(u16, i); }
```

Для работы с фреймами вводится специальная переменная FC. Когда создается новый фрейм, переменная FC увеличивается на 1, когда происходит возврат из фрейма, переменная уменьшается на 1. Переменная FC объявлена через ключевое слово `reg`, так как на текущий момент специального вида глобальных переменных не введено в nML. Объявление переменной FC на nML представлено ниже:

```
reg FC[FRAME_TYPE]
```

Для отслеживания текущего адреса инструкции используется счетчика инструкций (PC):

```
reg PC [i64]
```

Ниже приведено объявление памяти на nML для BM Ark:

```
shared mem MEM [MEM_SIZE, MEMORY_TYPE]
mem VM_MEM[VM_MEM_SIZE, MEMORY_TYPE]
```

Память `MEM` используется для хранения инструкций, и именно ее адреса ячеек хранит в себе `PC`. Память `VM_MEM` используется для хранения метаданных и экземпляров объектов.

Спецификация семантики инструкции виртуальной машины представляет собой процедуру и описывается на императивном языке, с помощью следующих элементов:

- Оператора присваивания;
- Управляющих конструкций;
- Рекурсивного поиска с заданными условиями;
- Составного оператора.

Управляющие конструкции представлены оператором с условием (if-then-else) и оператором возбуждения исключения. Управляющие конструкции и оператор присваивания используют набор битовых операции. Рекурсивный поиск с заданными условиями используется для работы с хранилищем метаданных. Составной оператор состоит из любого количества и любых комбинаций операторов присваивания, управляющих конструкций и рекурсивного поиска. Необходимыми типами данных для описания являются: битовые вектора произвольной длины и массивы. Для удобства описания также используется следующие типы: числа с плавающей точкой, списки и структуры. Стоит отметить, что вспомогательные типы можно смоделировать с помощью битовых векторов и массивов.

В описаниях инструкций управляющие регистры представлены глобальными переменными.

Фреймы представлены списком структур, содержащих атрибуты переменных и хранимые значения.

Ниже приведен пример спецификации локальной инструкции “div” на языке nML:

```
op div (vs1: V4, vs2: V4)
  init   = {image_size = OP_V4_V4_SIZE;}
  syntax = format("div %s, %s", vs1.syntax, vs2.syntax)
  image  = format("%s", op_v4_v4(DIV_V4_V4_OPCODE, vs1, vs2).image)
  action = {
    check_register_i32(vs1).action;
    check_register_i32(vs2).action;

    if (vs2 == 0) then
      add_exception(ArithmeticException).action;
    endif;

    if (vs1 == INT32_MIN && vs2 == -1) then
      acc_reg = coerce(i32, INT32_MIN);
    else
      acc_reg = vs1 / vs2;
    endif;

    set_acc_type_i32().action;
    set_acc_mask32().action;
  }
```

На листинге выше семантика инструкции **div** описана в атрибуте “`action`”, синтаксис описан в атрибуте “`syntax`”, в атрибуте “`image`” описана кодировка данной инструкции.

При возникновении исключения внутри инструкции, его обработка начинается с вызова метода `add_exception`, который передает управление конструктору класса исключения, где создается экземпляр исключения. При возбуждении исключения с помощью специальной инструкции `throw` на вход ей передается идентификатор экземпляра класса исключения. После выброса исключения начинается проверка, которая должна определить: находится ли инструкция, вызвавшая исключение, в try блоке. Если инструкция содержится в try блоке и исключение может быть обработано, осуществляется переход в catch блок, иначе происходит завершение исполнения метода и происходит проверка наличия подходящих try/catch блоков для инструкции, вызвавшей метод. Так будет происходить пока исключение не будет перехвачено или пока не будут завершены все методы.

Пример nML спецификации обработки выброса исключения представлен ниже. Приводится код внутренней операции `throw_recursion`, которая проверяет существование обработчика исключения для текущего метода, на адрес которого можно перейти.

```
internal op throw_recursion (class_id: MEMORY_TYPE)
  action = {
    this_method_id = frame_method_stack[FC];
    try_address   =   find_try_catch(TC_INDEX,   coerce(MEMORY_TYPE,
this_method_id), op_class_id, coerce(MEMORY_TYPE, CURRENT_PC));
    memory_get_element(try_address, EXISTENCE_FLAG).action;
    if (return_memory_element != 0) then
      // Найден подходящий обработчик исключений
      memory_get_element(try_address, HANDLER_PC_INDEX).action;
      temp1d1mt = return_memory_element;
```

```
        // Переход на catch блок
        PC = coerce(i64, temp1d1mt);
      else // Не найден обработчик исключений
        if (FC_VAR != 0) then // Не последний фрейм
          return_().action;
          CURRENT_PC = PC;
          // Флаг обозначающий повторный вызов этого метода
          return3mt = 1;
        else // Последний фрейм
          terminate_program = EXCEPTION_TERMINATE;
          if (main_terminate_address != 0) then
            PC = main_terminate_address;
          endif;
        endif;
      endif;
    }
```

В приведенном фрагменте кода операция `find_try_catch(...)` осуществляет рекурсивный поиск в хранилище метаданных обработчиков исключений с заданными параметрами. Листинг операции приведен ниже:

```
function find_try_catch(index: MEMORY_TYPE, value: MEMORY_TYPE,
                        id_class: MEMORY_TYPE, this_pc: MEMORY_TYPE)
                                : MEMORY_TYPE =
  if      VM_MEM[index + EXISTENCE_FLAG] != 0
    && !( VM_MEM[index + METHOD_ID_INDEX] == value
    && (VM_MEM[index + CATCH_CLASS_INDEX] == id_class ||
        VM_MEM[index + CATCH_CLASS_INDEX] == 0)
    && (VM_MEM[index + START_PC_INDEX] <= this_pc) &&
       (VM_MEM[index +   END_PC_INDEX] >= this_pc))
  then find_try_catch(index + TC_STR_SIZE, value, id_class, this_pc)
  else index
  endif
```

Операция `memory_get_element()` осуществляет чтение данных из памяти по заданному адресу и смещению. Листинг операции приведен ниже:

```
internal op memory_get_element(address: MEMORY_TYPE,
                               element: MEMORY_TYPE) action = {
  return_memory_element = VM_MEM[address + element];
}
```

На текущий момент спецификация системы команд на языке nML имеет ряд ограничений, связанных с ограничениями языка. Например, в текущей реализации глубина фреймов не может превышать 8.

3.2 Генерация тестовых программ

В предлагаемом методе функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд тестовые программы генерируются на основе тестовых шаблонов и исполнимой модели. Исполнимая модель генерируется на основе спецификации системы команд из раздела 3.1.

Для создания тестовых шаблонов в предложенном методе используются следующие подходы:

- Ручная разработка тестовых шаблонов.
- Автоматическая генерация тестовых шаблонов.
- Использование сторонних генераторов.

В реализации метода для описания тестовых шаблонов используется DSL Ruby. Применение такого подхода обусловлено несколькими преимуществами [31], а именно: позволяет использовать байт-код специфицированной архитектуры, при описании методов; позволяет использовать встроенные техники генерации и генераторов данных; позволяет использовать возможности языка Ruby.

Спецификации системы команд были расширены специальными псевдо (`pseudo`) операциями для возможности загрузки метаданных, описывающих классы, поля классов, методы, обработчики исключений. Данные операции также доступны для использования в шаблонах.

Для представления и работы с метаданными в шаблоны были добавлены специальные структуры для классов, полей классов, методов и try/catch блоков. Пример такой структуры для поля класса приведен на листинге ниже:

```
Field = Struct.new(:id, :type, :class_id, :access_flag, :name)
```

При описании метода в шаблоне, помимо его описания с помощью соответствующей структуры, создается блок кода, содержащий байт-код метода. Адрес начала блока кода хранится в структуре описывающий этот метод.

Приведем пример описания простого класса:

```
def initialize_x_class()
  @@class_00 = Class.new get_new_class_id(), C_M_COUNT, C_F_COUNT
  @@f_00 = Field.new get_new_field_id(), FIELD_TYPE_I32,
      @@class_00.id, ACC_PUBLIC + ACC_STATIC, ""
  @@m_00 = Method.new get_new_method_id(), @@class_00.id,
      ACC_PUBLIC + ACC_STATIC, "", M_P_SHIFT, M_P_COUNT, M_ADDRESS
end
```

Такое описание позволяет создавать классы с нужным количеством полей определенных типов и режимов доступа, задавать контракты методов и описывать try/catch блоки соответствующих методов. Такое описание метаданных используется во всех способах создания тестовых шаблонов.

3.2.1 Ручная разработка тестовых шаблонов

Ручная разработка тестовых шаблонов применяется для начальной настройки базовых шаблонов и для описания шаблонов, которые сложно получить другими способами, отражающие специфику системы. В качестве иллюстрации начальной настройки приведем объявления регистров:

```
# Defines alias methods for V8 registers
(0..255).each do |i|
  define_method "v8_#{i}" do |&contents| V8(i, &contents) end
end
```

Описание препаратора для регистра V8:

```
preparator(:target => 'V8', :mask => "XXXX") {
  movi_v8_imm16 target, value(0, 15)
}
```

Препаратор описывает как загрузить данные в регистр ВМ, это используется, если регистр не был ранее инициализирован.

За счет использования встроенных генераторов и ранее описанных или сгенерированных элементов тестовых шаблонов данный подход становится менее трудозатратным и более удобным по сравнению с обычной ручной разработкой тестовых программ.

3.2.2 Автоматическая генерация тестовых шаблонов

Автоматическая генерация тестовых шаблонов позволяет получить шаблоны для создание тестовых программ, соответствующих определенным критериям тестового покрытия [32]. Метод автоматической генерации тестового шаблона на основе спецификации системы команд состоит из следующих шагов:

Шаг 1. Создаем абстрактный шаблон, описывающий структуру генерируемого тестового шаблона. В качестве элементов абстрактного шаблона могут использоваться наборы инструкций и генераторы данных.

Шаг 2. Получаем набор инструкций системы команд, соответствующие определенным критериям, из спецификаций системы команд.

Шаг 3. Определяем специализированные генераторы данных.

Шаг 4. На основе абстрактного шаблона, генераторов данных и набора инструкций генерируем тестовые шаблоны.

В качестве примера абстрактного шаблона рассмотрим структуру, состоящую из блока тестового воздействия и блока подготовки данных. Блок тестового воздействия состоит из одной инструкции из системы команд; в блоке подготовки данных используются препараторы, инициализирующие входные параметры инструкции данными, подготавливаемыми генератором данных в соответствии с заданным критерием тестового покрытия.

Рассмотрим в качестве критерия тестового покрытия, покрытие всех достижимых путей в графе потока управления инструкций. Для достижения этого тестового покрытия можно использовать специальный генератор данных, нацеленный на покрытие всех достижимых путей исполнения инструкции. Этот генератор реализован в инструменте MicroTESK и его можно использовать в тестовом шаблоне следующим образом:

```
sequence {
  div v4(_), v4(_) do testdata('all_paths') end
}.run
```

Генератор данных “all_paths”, нацеленный на покрытие всех достижимых путей исполнения инструкции, устроен следующим образом. На основе спецификации системы команд на языке nML строится внутреннее представление описываемых инструкций. Для каждой инструкции строится граф потока управления и соответствующая SSA-форма. Для каждого пути в графе на основе SSA-формы строится условие прохождения по данному пути: логическое выражение, использующее параметры процедуры и глобальное состояние модели. Затем проводится трансляция полученных условий в язык ограничений SMT-LIB. Для этого требуется представить глобальное состояние модели и непосредственно условия в терминах SMT-LIB. Далее проводится построение ограничений для SMT-решателя, с помощью которых можно сгенерировать входные данные для заданного условия пути. Подробнее про построение SMT-LIB формул можно прочитать в работе [33].

Стоит отметить, что на текущий момент реализация генератора “all_paths” не отслеживает зависимости по данным, считываемым из памяти, из-за чего для инструкций, работающих с метаданными, часть тестовых шаблонов была описана в полуавтоматическом режиме. Но это ограничение реализации, а не метода.

3.2.3 Использование сторонних генераторов

Использование сторонних генераторов позволяет получать тестовые шаблоны от других специализированных инструментов. Для этого можно применить метод создания архитектурно независимых тестовых шаблонов для виртуальных машин и микропроцессоров [34]. Метод состоит из следующих шагов:

Шаг 1. Формируем набор базовых конструкций, на основе анализа сценария тестового шаблона и архитектурно зависимых элементов, которые в нем используются. Базовые конструкции используются для замены архитектурно зависимых элементов.

Шаг 2. Создаем архитектурно независимый шаблон, используя сторонний генератор для получения тестового сценария, использующего базовых конструкций.

Шаг 3. Описываем реализации базовых конструкций. Реализация описывается с использованием системы команд, информация о которой может быть получена из спецификации системы команд. При этом реализация конструкций может быть описана несколькими способами, в зависимости от богатства системы команд. Для выбора используемых реализаций могут быть использованы различные стратегии перебора, например, случайный выбор или полный перебор.

В данной работе при реализации метода были выделены следующие базовые конструкции: присваивания, сложения, цикла с параметрами, условного оператора, операции завершения выполнения цикла. В качестве примера можно привести конструкцию условного оператора:

```
def if_condition_op(b, if_condition, if_label)
  if if_condition == JNE then
    jne_v8_imm16 V8(b), if_label
  elsif if_condition == JEQ then
…
    jne_v8_imm16 V8(b), if_label
  end
end
```

Для генерации архитектурно независимых шаблонов использовался внешний генератор для проверки оптимизаций, реализованных в ВМ. Принцип работы этого генератора описан в работе [35].

3.3 Проверка результата исполнения тестовой программы

После исполнения тестовой программы на ВМ, необходимо проверить полученный результат. В данной работе рассматривается два метода проверки результатов тестовой программы: использование встроенных самопроверок (self-checks) в тестовых программах и проверка трассы исполнения тестовой программы с помощью оракула. Для обоих методов используется симулятор, который строится по формальным спецификациям системы команд ВМ.

Встроенные самопроверки – это код на языке тестируемой ВМ, который сравнивает состояние тестируемой системы с состоянием модели, вычисленным во время генерации тестовой программы. Код для самопроверок описывается заранее и называется компаратором (comparator). Компараторы вставляются в тестовую программу в места, где необходимо проверить значения памяти. Эталонное значение для сравнения берется из симулятора. Этот метод хорошо подходит для случаев, когда отсутствует трасса или способы ее проверить. Недостатком является дополнительный код, вносимый в тестовые программы, отсутствие точной локализации места ошибки и отсутствие проверки синтаксиса и кодировок инструкций.

Оракул – это инструмент, который проверяет (объясняет) трассу исполнения тестовой программы путем сравнения ее событий с эталонными событиями, получаемыми при

исполнении тестовой программы на симуляторе. События трассы исполнения – это события, возникшие в процессе исполнения теста.

3.3.1 Метод автоматизированного построения оракула

Предлагаемый метод основан на использовании спецификации системы команд для получения декодера инструкций и исполнимой модели. Метод автоматизированного построения оракула состоит из следующих шагов:

Шаг 1. Формулируем список событий трассы исполнения, проверка которых должна осуществляться оракулом. Типами таких событий могут быть исполнение инструкции и вывод информации о текущем состоянии (память, регистры, стек).

Шаг 2. Описываем регулярные выражения для распознавания событий трассы.

Шаг 3. На основе регулярных выражений конструируется адаптер трасс. Результатом работы адаптера трасс является список событий трассы в одном из форматов для обмена данными, например, в формате *.JSON.

Шаг 4. Создаем инструмент (парсер) для извлечения метаданных и списка инструкций метода из исполнимого файла тестовой программы.

Шаг 5. На основе спецификации системы команд получаем декодер инструкций.

Шаг 6. На основе спецификации системы команд получаем исполнимую модель.

Шаг 7. Создаем механизм загрузки метаданных и декодированных инструкций метода в память модели.

Шаг 8. Связываем события трассы с событиями в исполнимой модели.

Шаг 9. Формулируем критерии сравнения событий.

Шаг 10. Реализуем алгоритмы симуляции тестовой программы и проверки событий. Например, может быть использовано несколько стратегий: для проверки трасс с использованием JIT оптимизаций и без них.

Полученный таким способом оракул принимает на вход трассу исполнения и исполнимый файл тестовой программы. Результатом работы оракула должен быть вердикт о соответствии трассы и тестовой программы. В зависимости от реализации может выводиться дополнительная информация, например, об различиях в событиях трассы и симулятора, или об отсутствующих инструкций в трассе.

3.3.2 Схема работы оракула

Опишем подробнее схему проверки трассы исполнения тестовой программы с помощью оракула:

Шаг 1. Средствами ВМ проверяем тестовую программу на корректность с помощью верификатора и получаем бинарную программу. Бинарная программа является исполнимым файлом и может быть выполнена на ВМ.

Шаг 2. Исполняем бинарную тестовую программу на ВМ и получаем трассу исполнения.

Шаг 3. Получаем метаинформацию и байт-код методов. Для этого подаем на вход парсеру бинарную программу, полученную ранее.

Шаг 4. Собираем события трассы. Для этого используем адаптер трасс, на выходе получаем список событий в формате JSON.

Шаг 5. Получаем результат проверки трассы тестовой программы. На вход оракул принимает: события трассы исполнения в JSON формате, метаданные программы и байт-код методов. На основе этих данных оракул выносит вердикт о соответствии или несоответствии полученной трассы поданной тестовой программе.

Данная схема представлена на рис. 3. Детали реализации алгоритма работы оракула описаны в следующем подразделе.

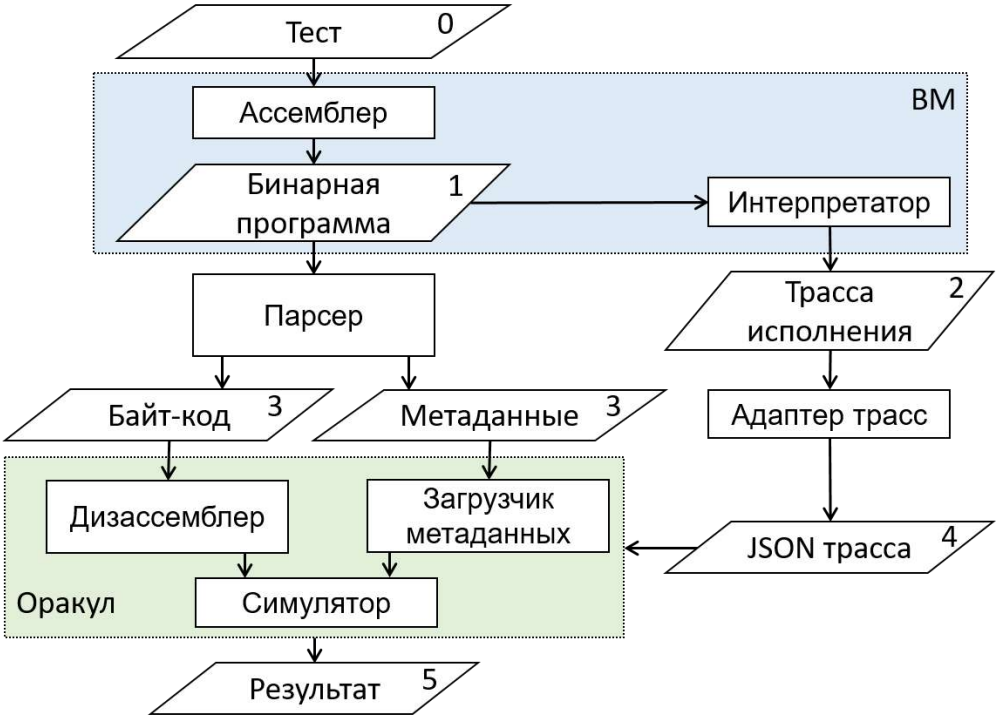


Рис. 3. Схема проверки трассы с помощью оракула.
Fig. 3. Scheme for checking the test program trace using oracle.

3.3.3 Пример работы оракула

Основные элементы оракула: симулятор и дизассемблер строятся автоматически инструментом MicroTESK на основе формальных спецификаций на nML. С помощью дизассемблера распознается байт-код методов из бинарной программы. Симулятор используется для исполнения байт-кода. Загрузка метаданных осуществляется через специальный модуль инструмента, позволяющий загружать метаданные в память симулятора. Для симуляции программы оракулу необходима вся информация об используемых классах. Если в программе используются библиотечные классы, то информацию о них необходимо подготовить и загрузить в симулятор.

Для анализа трассы исполнения от ВМ применяется адаптер трасс. Ниже приведен пример фрагмента трассы irtoc интерпретатора ВМ Ark:

```
[TID 031b4e] D/interpreter:  pc: 0x7f30ca8bf173 ---> movi v0, 118
[TID 031b4e] D/interpreter:   acc.pri = (i64) 0 | (f32) 0
                                      | (f64) 0 | (hex) 0
[TID 031b4e] D/interpreter:   v0.pri = (i64) 118 | (f32) 1.65353e-43
                                     | (f64) 5.82997e-322 | (hex) 76
```

Для анализа событий адаптер трасс использует регулярные выражения, что позволяет его настраивать на различные виды трасс. С помощью адаптера трасс собираются события из трассы и представляются в специальном JSON формате, что позволяет сравнивать их с исполняемым в симуляторе байт-кодом и состоянием симулятора. Ниже приведен пример распознанных событий в формате JSON трассы: исполнение инструкции и вывод информации о текущем состоянии (памяти):

```
{"EventType": "INSTRUCTION", "Instruction": "movi v0, 118",
              "ThreadIdentifier": "031b4e", "PC": "7f30ca8bf173"},
{"EventType": "MEMORY", "ThreadIdentifier": "031b4e", "RegisterType":
              "pri", "RegisterValue": "0", "Register": "acc"},
{"EventType": "MEMORY", "ThreadIdentifier": "031b4e", "RegisterType":
              "pri", "RegisterValue": "76", "Register": "v0"}
```

Оракул читает событие из трассы: если это состояние, то он проверяет совпадает ли оно с состоянием в симуляторе; если это инструкция, то оракул сравнивает ее со следующей инструкцией в очереди и исполняет ее. Если оракул работает в режиме полного совпадения, то любое расхождение является признаком несоответствия трассы программе. Ниже приведен пример работы оракул для сравнения событий:

```
Compare Instruction: [SUCCESS],
                     (trace : model) -> 'movi v0, 118' : 'movi v0, 118'
[ProgramSimulator] PC: 0x64, call: movi v0, 118
Compare Registers: [SUCCESS], Mask: 0x0000000000000000,
                     (trace : model) -> 'acc' = 0x0 : 'ACC_G' = 0x0
Compare Registers: [SUCCESS], Mask: 0x00000000FFFFFFFF,
                     (trace : model) -> 'v0' = 0x76 : 'R[0]' = 0x76
```

Оракул в режиме с пропусками, в случае расхождения, начинает перебирать следующие инструкции и состояния для сравнения. Признаком успешности для него является нахождение соответствия всем события в трассе, при этом он выводит список инструкций, которых в трассе обнаружено не было. Такой режим оракула, в отличии от простого сравнения трасс, применим для проверки трасс, полученных после JIT- и AOT- компиляции, дающих трассы с пропусками.

4. Полученные результаты

Описываемый в работе метод функционального тестирования языковых ВМ на основе формальных спецификаций применялся для тестирования ВМ Ark. При тестировании ВМ были найдены несоответствия в спецификации системы команд и ошибки в реализации некоторых механизмов ВМ.

На момент написания статьи на языке nML было специфицировано 298 инструкций/байт-код команд ВМ Ark. Подробнее информация по группам инструкций представлена в табл. 1.

Табл. 1. Статистика по специфицированным инструкциям виртуальной машины Ark.
Table 1. Statistics for Ark VM specified instructions.

Группа инструкций	Кол-во	Спец.
Арифметические	131	131
Управляющие	35	35
Арифметические для чисел с плавающей точкой	34	34
Работы с регистрами	32	32
Работы с памятью/Системные	42	42
Динамические	6	6
Работы с массивами	18	18
Всего	298	298

Сгенерированный тестовый набор для проверки работы отдельных инструкций ВМ содержит 496 тестовых программ. Покрытие кода ‘cpp’ интерпретатора ВМ данным тестовым набором составило 85%. Измерение покрытия кода проводилось с помощью инструмента **gcov** [36]. Отчасти такой процент покрытия можно объяснить тем, что интерпретатор содержит код, для обработки ситуаций отсутствия в исполняемом файле классов, методов и пр., который покрыт не был.

С помощью данного тестового набора было найдено несколько ошибок. Отсутствовали реализации 2х инструкций, описанных в документации, и отсутствовала возможность корректного вызова еще 3х инструкций. Найдены мелкие ошибки в синтаксисе инструкций в трассе исполнения программы. С помощью этого же тестового набора было найдено расхождение поведения верификатора для однотипных инструкций, в которых присутствовала обработка ситуации, при подаче на вход null объекта. В одном случае верификатор не сообщал о проблемах, во втором случае сообщал об ошибке с сообщением о невозможности подачи на вход null объекта.

Были найдены ошибки с помощью тестовых программ, полученных из шаблонов, написанных вручную. Одна из таких ошибок связана с использованием try/catch блоков. Пример метода такой тестовой программы приведен ниже:

```
.function i64 class10.method0(i64 a0, i64 a1) {
  nop
try_start_instruction:
  nop
# Preparation
  ldai.64 0
# Stimulus
  not.64
try_end_instruction:

  nop
  ststatic.64 class10.field0
catch_instruction_block_begin:
  return.64

  .catchall try_start_instruction, try_end_instruction,
catch_instruction_block_begin
}
```

Такая программа успешно проходила проверку верификатором ВМ Ark, успешно исполнялась интерпретаторами ВМ и соответствовала предоставленной на тот момент документации. Но при попытке применения к методу JIT или AOT компиляторов выдавалась ошибка. Проблема продемонстрированного метода заключалась в том, что в качестве выхода из метода используется инструкция return.64 из блока обработки исключения. Стоит особенно отметить, что сложно представить, как такой тестовый пример можно получить с языка высокого уровня, что подтверждает полезность методов генерации тестовых программ на уровне байт-кода.

Еще одна интересная ошибка была найдена тестовой программой, полученной из шаблона, разработанного вручную, для проверки работы регистров с максимальными индексами в методе. Максимально возможное значение индекс регистра в методе ВМ Ark равно 65535 (2^16). Но передаваемые в метод параметры загружаются в регистры, следующие за максимальным используемом в методе. И при использовании в методе регистра с максимальным индексом 65535, передаваемые параметры загружались в регистры с индексами 65536, 65537, что являлось ошибкой.

5. Дальнейшие исследования

На основе опыта применения метода и полученных результатов можно сформулировать следующие идеи и направления для дальнейшего развития подхода и прототипа:

- Автоматизация создания тестовых наборов на основе генерации тестовых последовательностей из 2-х и более взаимодействующих инструкций. При тестировании микропроцессоров большое количество ошибок наблюдается при выполнении цепочек инструкций [22, 37-38] на тестируемом устройстве. Предлагаемый подход можно расширить генераторами и шаблонами, нацеленными на покрытие исполнения цепочек инструкций.
- Расширение набора автоматически генерируемых тестовых шаблонов и модели ошибок для них.
- Автоматизированное создание препараторов и компараторов для тестовых шаблонов. В подходе при описании шаблонов, некоторые части по-прежнему приходится описывать вручную. К таким частям можно отнести: препараторы, необходимые для инициализации локальной памяти (регистров) и компараторов, используемых при проверках состояния локальной памяти.
- Развитие языка nML. На текущий момент спецификация ISA BM Ark на nML имеет ряд ограничений, например, максимальное количество доступных фреймов ограничено 8.
- Расширение списка генераторов данных и структурных генераторов, используемых в инструменте MicroTESK.

6. Заключение

В данной статье предложен метод функционального тестирования языковых ВМ на основе формальных спецификаций. В работе было показана возможность написания спецификаций на языке описания архитектуры nML для системы команд ВМ. В работе показано применение тестовых шаблонов с использованием метаданных, которые позволяют создавать классы с заданными требованиями. В описанном подходе основным способом воздействия на ВМ являются программы на байт-коде, что позволяет воздействовать на всю функциональность ВМ. Было показано, как на основе формальных спецификаций можно получить тесты для покрытия всех достижимых путей исполнения инструкции. Был описан метод, позволяющий использовать сторонние генераторы для получения тестов, нацеленных на проверку корректности оптимизации и предложена концепция архитектурно независимых шаблонов. Для проверки результатов исполнения тестовых программ в подходе используется оракул, создаваемый на основе спецификаций системы команд. Оракул позволяет проверять обычные трассы и трассы с пропусками, полученные при использовании JIT- и AOT-компиляции. Данный подход был применен к тестированию ВМ Ark во время ее разработки. Было получено высокое (85%) покрытие кода интерпретатора, найдены ошибки в спецификации системы команд и в реализации ВМ. Метод показал свою полезность при функциональном тестировании ВМ и имеет перспективы развития.

Список литературы / References

[1]. Smith J., Nair R. Virtual machines: versatile platforms for systems and processes. – Elsevier, 2005.
[2]. Li X. F. Advanced design and implementation of virtual machines. – CRC Press, 2016.
[3]. Amdahl G. M., Blaauw G. A., Brooks F. P. Architecture of the IBM System/360 //IBM Journal of Research and Development. – 1964. – Т. 8. – №. 2. – С. 87-101.
[4]. Howden W. E. Theoretical and empirical studies of program testing //IEEE Transactions on Software Engineering. – 1978. – №. 4. – С. 293-298.

[5]. Open-source проект Ark (Panda) Runtime [Электронный ресурс]. – Режим доступа: https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/static_core/
[6]. Polito G., Ducasse S., Tesone P. Interpreter-guided differential JIT compiler unit testing //Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. – 2022. – С. 981-992.
[7]. Sirer E. G., Bershad B. N. Using production grammars in software testing //ACM SIGPLAN Notices. – 1999. – Т. 35. – №. 1. – С. 1-13.
[8]. Java Language and Virtual Machine Specifications [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/javase/specs/
[9]. Sirer E. G., Bershad B. N. Testing Java virtual machines //Proc. Int. Conf. on Software Testing And Review. – 1999.
[10]. Chen Y. et al. Coverage-directed differential testing of JVM implementations //proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. – 2016. – С. 85-99.
[11]. Chen Y., Su T., Su Z. Deep differential testing of JVM implementations //2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). – IEEE, 2019. – С. 1257-1268.
[12]. Zhao Y. et al. History-Driven Test Program Synthesis for JVM Testing. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). 1133–1144 [Электронный ресурс]. https://doi.org/10.1145/3510003.3510059
[13]. Wu M. et al. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 56–68 [Электронный ресурс]. DOI: 10.1109/ICSE48619.2023.00017
[14]. Wu M. et al. SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing //Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. – 2023. – С. 1062-1074.
[15]. Zang Z. et al. Compiler testing using template java programs //Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. – 2022. – С. 1-13.
[16]. Calvagna A., Tramontana E. Automated conformance testing of Java virtual machines //2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems. – IEEE, 2013. – С. 547- 552.
[17]. Misse-Chanabier P. Testing a virtual machine developed in a simulation-based virtual machine generator: дис. – Université de Lille, 2022.
[18]. Hrishikesh M.S., Rajagopalan M., Sriram S., Mantri R. System Validation at ARM — Enabling our Partners to Build Better Systems. White Paper. April 2016
[19]. Mahapatra R.N., Bhojwani P., Lee J., and Kim Y. Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No.43 Phase 3 Report. 2009. 43 P.
[20]. Adir A. et al. Genesys-pro: Innovations in test program generation for functional processor verification //IEEE Design & Test of Computers. – 2004. – Т. 21. – №. 2. – С. 84-93.
[21]. Li T. et al. MA/sup 2/TG: a functional test program generator for microprocessor verification //8th Euromicro Conference on Digital System Design (DSD'05). – IEEE, 2005. – С. 176-183.
[22]. Камкин А. С. Генерация тестовых программ для микропроцессоров //Труды Института системного программирования РАН. – 2008. – Т. 14. – №. 2. – С. 23-63.
[23]. Kamkin A. Combinatorial model-based test program generation for microprocessors //Preprint of ISPRAS. – 2009.
[24]. Chupilko M., Kamkin A., Protsenko A., Smolov S., Tatarnikov A. MicroTESK: Automated Architecture Validation Suite Generator for Microprocessors. DVCon Europe 2018.
[25]. Kamkin A., Tatarnikov A. MicroTESK: A Tool for Constrained Random Test Program Generation for Microprocessors //Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11. – Springer International Publishing, 2018. – С. 387-393.
[26]. Peña R. et al. SMT-Based Test-Case Generation and Validation for Programs with Complex Specifications //Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems: Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday. – Cham: Springer Nature Switzerland, 2023. – С. 188-205.
[27]. Khurshid S., Marinov D. TestEra: Specification-based testing of Java programs using SAT //Automated Software Engineering. – 2004. – Т. 11. – С. 403-434.
[28]. Open-source project MicroTESK [Электронный ресурс]. – Режим доступа: https://forge.ispras.ru/projects/microtesk

[29]. Freericks M. The nML machine description formalism. – Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4, 1991.
[30]. Vishnoi S. K. Functional simulation using sim-nml //Master's thesis, Department of Computer Science and Engineering, IIT Kanpur. – 2006.
[31]. Татарников А.Д. Автоматизация конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций. Диссертация на соискание ученой степени к.т.н. Институт системного программирования РАН, Москва, 2017. 162 с.
[32]. Проценко А. С., Татарников А. Д. Автоматическая генерация тестовых шаблонов на основе спецификаций системы команд //Новые информационные технологии в исследовании сложных структур. – 2018. – С. 82-83.
[33]. Kamkin A., Khoroshilov A., Kotsynyak A., Putro P. Deductive binary code verification against source-code-level specifications //Tests and Proofs: 14th International Conference, TAP 2020, Held as Part of STAF 2020, Bergen, Norway, June 22–23, 2020, Proceedings 14. – Springer International Publishing, 2020. – С. 43 - 58.
[34]. Проценко А. С. Архитектурно независимые тестовые шаблоны для виртуальных машин и микропроцессоров. ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ, Том 30, № 11, 2024, С. 579–584. doi:10.17587/it.30.579-584
[35]. Zelenov S., Zelenova S. Model-based testing of optimizing compilers //International Workshop on Formal Approaches to Software Testing. – Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. – С. 365-377.
[36]. gcov – a Test Coverage Program [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[37]. TMS570LS31x/21x Series Microcontroller, Silicon Errata. Texas Instruments, 2013. https://www.ti.com/lit/er/spnz195g/spnz195g.pdf
[38]. dsPIC30F5011/5013 Rev A1/A2 Silicon Errata. Microchip, 2005. https://ww1.microchip.com/downloads/en/DeviceDoc/80210e.pdf

Информация об авторах / Information about authors

Александр Сергеевич ПРОЦЕНКО является научным сотрудником отдела технологий программирования ИСП РАН. Область научных интересов: языковые виртуальные машины, микропроцессоры, архитектура системы команд, верификация и тестирование.

Alexander Sergeevich PROTSENKO is a researcher at the Software Engineering Department of ISP RAS. His research interests include language virtual machines, microprocessors, instruction set architecture, verification and testing.

DOI: 10.15514/ISPRAS-2025-37(1)-5

CC BY

Подход к построению компиляторов нейронных сетей с использованием инфраструктуры MLIR

[1,2] *И.И. Кулагин, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,3] *А.В. Вязовцев, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>*
[1,2] *М.М. Романов, ORCID: 0009-0001-3242-1171 <mmromanov@ispras.ru>*
[1,2] *Д.М. Мельник, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[3] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9*

Аннотация. Развитие матричных расширений процессорных архитектур, а также внедрение этих расширений в специализированные AI-процессоры, позволяет существенно повысить эффективность выполнения искусственных нейронных сетей. В работе выполнен обзор базовых функциональных возможностей некоторых популярных матричных расширений процессорных архитектур, в частности расширений ARM SME, RISC-V IME, RISC-V AME, а также процессорной архитектуры DaVinci. В результате проведенного анализа была предложена модель абстрактного матричного процессора, отражающая особенности современных процессорных архитектур, которые поддерживают матричное расширение. Для введенной модели матричного процессора разработано гетерогенное матричное промежуточное представление, которое может быть использовано для построения компиляторов нейронных сетей. Предложенное промежуточное представление было реализовано в инфраструктуре MLIR в виде диалекта heteroMx. В работе также описан подход к построению AI-компилятора с использованием разработанного диалекта heteroMx. Разработанное промежуточное представление может быть адаптировано или конкретизировано для других матричных процессорных архитектур.

Ключевые слова: матричное расширение; архитектуры RISC-V; расширенная архитектура матричных вычислений ARM SME; промежуточное представление; инфраструктура MLIR.

Для цитирования: Кулагин И.И., Бучацкий Р.А., Пантилимонов М.В., Вязовцев А.В., Романов М.М., Мельник Д.М. Подход к построению компиляторов нейронных сетей с использованием инфраструктуры MLIR. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 87–106. DOI: 10.15514/ISPRAS–2025–37(1)–5.

Approach to Building AI-Compilers Using the MLIR Framework

[1,2] *I.I. Kulagin, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,3] *A.V. Vyazovtsev, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>*
[1,2] *M. M. Romanov, ORCID: 0009-0001-3242-1171 <mmromanov@ispras.ru>*
[1,2] *D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.

Abstract. The development of matrix extensions of processor architectures, as well as the implementation of these extensions in specialized AI processors, can significantly improve the efficiency of artificial neural networks. The paper provides an overview of the basic functionality of some popular matrix extensions of processor architectures, in particular, ARM SME, RISC-V IME, RISC-V AME extensions, as well as the DaVinci processor architecture. As a result of the analysis, a model of an abstract matrix processor was proposed. This model reflects the features of modern processor architectures supporting matrix extensions. For the introduced model of the matrix processor, a heterogeneous matrix intermediate representation was developed, which can be used to build compilers for neural networks. The proposed intermediate representation was implemented in the MLIR infrastructure as a heteroMx dialect. The paper also describes an approach to building an AI compiler using the heteroMx dialect. The developed intermediate representation can be adapted or specified for other matrix processor architectures.

Keywords: matrix extension; RISC-V; ARM SME; AI-compiler intermediate representation; MLIR infrastructure.

For citation: Kulagin I.I., Buchatskiy R.A., Pantilimonov M.V., Vyazovtsev A.V., Romanov M. M., Melnik D.M. Approach to Building AI-Compilers Using the MLIR Framework. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 87-106 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-5.

1. Введение

Значительная часть вычислений при выполнении искусственных нейронных сетей приходится на операции свертки и умножения матриц. Для ускорения этих операций в частности и всей нейронной сети вообще активно ведется разработка расширений процессорных архитектур, которые реализуют матричные инструкции. Среди наиболее известных матричных расширений можно выделить расширение Scalable Matrix Extension архитектуры ARM, расширения Integrated Matrix Extension и Attached Matrix Extension архитектуры RISC-V. Также можно отметить процессорную архитектуру DaVinci от компании Huawei, нейроморфный процессор NeuroMorphic Processor (NMP) [1] от компании LG и тензорный процессор от компании Google (Google TPU) [2] и др.

Матричное расширение процессорной архитектуры обычно содержит множество матричных регистров и специальные инструкции, которые реализуют умножение матриц. Матричное расширение может быть разработано на основе векторного расширения. В этом случае матричные регистры реализуются на основе регистрового файла векторного расширения, например, путем объединения части векторных регистров в матрицу или использования векторного регистра для хранения небольшого блока матрицы. Такое матричное расширение называется интегрированным. В случае, если матричное расширение не использует векторный регистровый файл, оно называется независимым. Процессоры, реализующие

интегрированное расширение, отличаются значительно меньшей стоимостью по сравнению с теми, которые реализуют независимое расширение.

Генерация эффективного кода для матричных процессорных архитектур проблематична в условиях отсутствия высокоуровневого промежуточного представления (Intermediate Representation – IR). Используемый в качестве промышленного стандарта LLVM IR не подходит в качестве такого IR, так как является слишком низкоуровневым. По этой причине в работе предложено гетерогенное матричное промежуточное представление.

Реализация предложенного промежуточного представления требует разработки объемной инфраструктуры, включающей в себя менеджер проходов, движок сопоставления с образцом, интерфейсы для трансляции инструкции одних промежуточных представлений в инструкции других и т.д. Одним из перспективных подходов к разработке промежуточных представлений является использование инфраструктуры проекта MLIR. Инфраструктура MLIR позволяет автоматизировать процесс разработки промежуточных представлений различных уровней. Предложенное гетерогенное матричное IR реализовано с использованием инфраструктуры MLIR [3].

Дальнейшее повествование в работе ведется по следующему плану. Во втором разделе будет выполнен обзор базовых возможностей существующих матричных расширений процессорных архитектур. После чего будет введена модель абстрактного матричного процессора. В третьем разделе будут раскрыты детали предложенного матричного промежуточного представления, выполнен короткий обзор инфраструктуры MLIR. Далее будет представлена функциональная структура AI-компилятора. В четвертом разделе содержатся выводы и обозначены некоторые перспективные направления для дальнейших исследований.

2. Модель архитектуры процессора с матричным расширением

В данном разделе представлена модель абстрактной архитектуры процессора, поддерживающая матричное расширение набора команд. Предлагаемая модель отражает некоторые аспекты существующих матричных расширений и процессорных архитектур таких, как матричное расширение ARM Scalable Matrix Extension (ARM SME), RISC-V Integrated Matrix Extension (RISC-V IME), RISC-V Attached Matrix Extension (RISC-V AME), Huawei Da Vinci. К числу таких аспектов можно отнести особенности реализации иерархической организации памяти, семантику инструкций матричного расширения и режимы работы с матричными регистрами.

2.1 Расширение ARM Scalable Matrix Extension

Расширение ARM Scalable Matrix Extension было включено в архитектуру ARM для A-профиля 20 марта 2024 года. SME [4] является продолжением развития масштабируемого векторного расширения Scalable Vector Extension (SVE) [5]. SVE добавляет в архитектурное состояние следующие компоненты: 32 масштабируемых векторных регистра (Z0-Z31), 16 предикатных регистров (P0-P15), специальный регистр first-fault register (FFR), управляющие регистры для различных режимов работы процессора (ZCR_EL1-ZCR_EL3). Масштабируемые векторные регистры не имеют фиксированной длины, каждая реализация процессора может содержать векторные регистры необходимой длины. Архитектура только накладывает ограничение по кратности на длину регистров. Она должна быть кратна 128 битам в диапазоне от 128 до 2048 бит. Масштабируемые регистры SVE расширения позволяют абстрагироваться от конкретной длины векторных регистров при разработке векторизованных программ, а компилятор генерирует код, содержащий универсальные инструкции, не зависящие от конкретной реализации архитектуры. Также SVE содержит множество инструкций: инструкции загрузки данных в масштабируемые регистры и

выгрузки из них в память, инструкции арифметических и логических операций над элементами векторных регистров, инструкции для спекулятивной векторизации и др.

Позже расширение SVE было расширено дополнительными инструкциями до SVE2. Были добавлены инструкции построения гистограмм, инструкции сопоставления с образцом, инструкции, реализующие криптографические алгоритмы, и др.

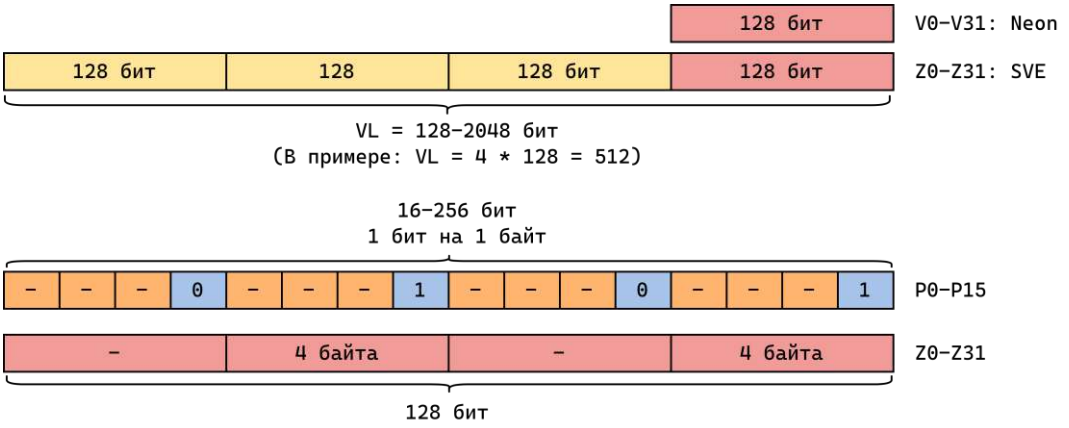


Рис. 1. Структура масштабируемых векторных регистров ARM SVE.
Fig. 1. The structure of ARM SVE scalable vector registers.

На рис. 1 представлена структура масштабируемых векторных регистров, а также масштабируемых предикатных регистров. Длина вектора неизвестна на этапе компиляции и может отличаться на различных реализациях архитектуры. В данном примере длина вектора равна 512 бит, т.е. 4 × 128 бит, что удовлетворяет ограничениям SVE, а масштаб векторного регистра (V scale) равен 4. Набор команд расширения SVE спроектирован таким образом, чтобы избежать в коде использования конкретной длины векторных регистров. Для этой цели расширение содержит 16 предикатных регистров. Предикатные регистры выступают в роли флага использования векторной инструкцией соответствующего элемента векторного регистра. Каждому биту предикатного регистра соответствует 8 бит векторного регистра. Таким образом, длина предикатных регистров в зависимости от реализации архитектуры может варьироваться от 16 до 256 бит.

На рис. 2 представлен пример вычисления SAXPY с использованием SVE инструкций. В этом примере особый интерес представляет инструкция whilelt. Эта инструкция инициализирует предикатный регистр, исходя из результата сравнения i < n, количества оставшихся для выполнения итераций с фактической длиной масштабируемых векторных регистров.

Расширение SME включает в себя SVE2 и определяет следующие ключевые возможности:

- масштабируемый двумерный ZA регистр (матричный регистр), позволяющий хранить двумерные матричные блоки;
- режим Streaming SVE (SSVE) – потоковый режим работы с векторными инструкциями расширения SVE;
- инструкции, вычисляющие внешнее произведение векторов в результирующий матричный регистр, используемый как аккумулятор;

инструкции загрузки, хранения и перемещения, которые передают вектор в строку или столбец матричного блока.

Регистр ZA используется матричными инструкциями как аккумулятор. Этот регистр состоит из виртуальных матричных блоков (tiles), количество и размер которых определяются

размером типа элементов. В табл. 1 представлены возможные конфигурации матричного регистра для различных типов элементов.

```c
void saxpy(float *x, float *y, float a, int n) {
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}
```

а) Реализация saxpy на языке C
a) Implementation of saxpy in C

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy:
  ldrsw x3, [x3]            // x3=*n
  mov x4, #0                // x4=i=0
  ldr s0, [x2]              // s0=*a
  b .latch
.loop:
  ldr s1, [x0, x4, lsl #2]  // s1=x[i]
  ldr s2, [x1, x4, lsl #2]  // s2=y[i]
  fmadd s2, s1, s0, s2      // s2+=x[i]*a
  str s2, [x1, x4, lsl #2]  // y[i]=s2
  add x4, x4, #1            // i+=1
.latch:
  cmp x4, x3                // i < n
  b.lt .loop                // more to do?
  ret
```

б) Скалярная реализация saxpy на ассемблере
b) Scalar implementation of saxpy in assembly

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy:
  ldrsw x3, [x3]          // x3=*n
  mov x4, #0              // x4=i=0
  whilelt p0.w, x4, x3    // p0=while(i++<n)
  ld1rw z0.w, p0/z, [x2]  // p0:z0=bcast(*a)
.loop:
  ld1w z1.w, p0/z, [x0, x4, lsl #3] // p0:z1=x[i]
  ld1w z2.w, p0/z, [x1, x4, lsl #3] // p0:z2=y[i]
  fmla z2.w, p0/m, z1.w, z0.w       // p0?z2+=x[i]*a
  st1w z2.w, p0, [x1, x4, lsl #3]   // p0?y[i]=z2
  incw x4                          // i+=(VL/32)
.latch:
  whilelt p0.w, x4, x3  // p0=while(i++<n)
  b.first .loop         // more to do?
  ret
```

в) Реализация saxpy
с использованием ARM SVE инструкций
c) Implementation of saxpy using ARM SVE instructions

Рис. 2. Пример реализации saxpy на языке C и ассемблере с использованием инструкций ARM SVE.
Fig. 2. Implementation of saxpy in C and assembly using ARM SVE instructions.

Табл. 1. Зависимость размера виртуальных блоков регистра ZA от типа элемента.
Table 1. Dependency of the size of the virtual blocks of the register ZA on the element type.

Тип элемента	Кол-во блоков (tiles)	Размер виртуального блока	Название виртуальных блоков	Размер регистра Z0-Z31
i8	1	(16 * vscale) × (16 * vscale)	ZA0.B	16 * vscale
i16	2	(8 * vscale) × (8 * vscale)	ZA0-ZA1.H	8 * vscale
i32/f32	4	(4 * vscale) × (4 * vscale)	ZA0-ZA3.S	4 * vscale
i64/f64	8	(2 * vscale) × (2 * vscale)	ZA0-ZA7.D	2 * vscale
i128	16	(1 * vscale) × (1 * vscale)	ZA0-ZA15.Q	1 * vscale

Физический размер матричного регистра ZA зависит от конкретной реализации архитектуры, как и в случае с масштабируемыми векторными регистрами. Спецификация SME лишь накладывает ограничения на размер матричного регистра. Его размер определяется длиной потокового вектора (Streaming Vector Length – SVL), то есть длиной масштабируемых регистров Z0-Z31 в потоковом режиме, и должен быть равен SVL × SVL. Таким образом, SME-код не оперирует конкретными размерами матричных блоков, так как их размер является неизвестным и может отличаться на различных реализациях архитектуры. Например, на процессоре Apple M4 длина потокового вектора SVL равна 512 бит, следовательно, данная реализация содержит 32 регистра Z0-Z31 длиной 512 / 8 = 64 байт, а также матричный регистр ZA размером 64 × 64 = 4096 байт. На рис. 3 представлен пример организации ZA регистра для случая, когда размер типа элемента равен 32 бита.

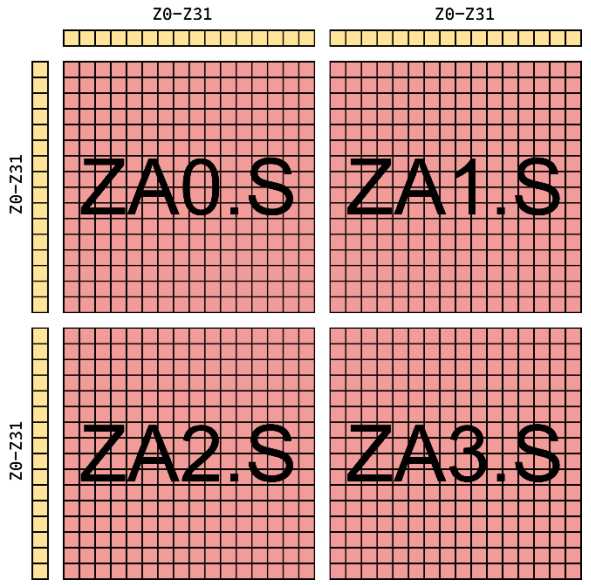


Рис. 3. Организация матричного регистра ZA. SVE = 512; типа элемента i32.
Fig. 3. Organization of the matrix register ZA with SVE = 512 and element type i32.

Ключевыми инструкциями SME являются инструкции внешнего произведения. Эти инструкции принимают в качестве входных операндов два масштабируемых SVE-регистра, хранящих вектора, и накапливают в матричном регистре ZA результат. Мнемоники инструкций внешнего произведения заканчиваются на MOPA. Например, инструкция внешнего произведения для чисел с плавающей запятой – FMOPA. Для типа FP32 инструкция FMOPA вычисляет внешнее произведение двух векторов размером SVL/32 элементов, расположенных в двух регистрах Z, и аккумулирует результат в регистре ZA размером SVL/32 × SVL/32 элементов. Поскольку у M4 SVL равна 512 битам, это означает, что инструкция FP32 FMOPA вычисляет внешнее произведение двух векторов по 16 элементов каждый и аккумулирует результат в регистре ZA размером 16 × 16 элементов. Таким образом, одна инструкция FP32 FMOPA на M4 выполняет 16 × 16 × 2 = 512 операций с плавающей запятой. Для выполнения инструкций SME необходимо переключить процессорное ядро в режим Streaming SVE. Активация этого режима осуществляется при помощи инструкции **SMSTART**, после чего архитектурное состояние SME становится доступным. Аналогично инструкция **SMSTOP** отключает режим SSVE. При переключении режимов происходит очистка регистров ZA и Z.

На рис. 4 приведен фрагмент SME-микроядра, выполняющего умножение матриц с элементами типа FP32 для одного блока (C += ABT). Матрицы A и C хранятся по столбцам, а матрица B по строкам. Фрагмент кода содержит только внутренний цикл, результат вычислений накапливается в регистре ZA. Так как для элементов FP32 регистр ZA состоит из 4 виртуальных блоков, то в результате вычисления данного цикла будет получен один блок матрицы C размером до (4 × SVL/32 × SVL/32) элементов. В случае выполнения данного фрагмента кода на процессоре Apple M4 блок матрицы C может быть размером до 32 × 32 элементов.

В теле цикла сперва загружается фрагмент столбца матрицы A (строка 5). Для процессора Apple M4 размер загруженного фрагмента может быть до 32 элементов. В случае если размер столбца меньше или не кратен 32 элементам, предикатный регистр PN8 обеспечит загрузку максимально возможного числа элементов. Далее, таким же образом загружаются элементы строки матрицы B (строка 6). Две инструкции в строках 7 и 8 инкрементируют адреса в X0 и

X1, указывая на следующий столбец A и следующую строку B. Инструкции FMOPA в строках 9-12 вычисляют четыре соответствующих внешних произведения и обновляют виртуальные матричные блоки ZA. После завершения цикла K (строки 3, 4 и 13) заканчивается вычисление блока C, и результат вычислений записывается из матричного регистра ZA в память. Далее вычисления будут повторяться для следующих блоков матрицы C.

```
1  // set predicate registers
2  // set register offset
3  k_loop:
4    sub x8, x8, #0x1
5    ld1w {z0.s, z1.s}, pn8/z, [x0]
6    ld1w {z2.s, z3.s}, pn9/z, [x1]
7    add x0, x0, x9
8    add x1, x1, x10
9    fmopa za0.s, p1/m, p0/m, z2.s, z0.s
10   fmopa za1.s, p1/m, p2/m, z2.s, z1.s
11   fmopa za2.s, p3/m, p0/m, z3.s, z0.s
12   fmopa za3.s, p3/m, p2/m, z3.s, z1.s
13 cbnz x8, k_loop
```

Рис. 4. Фрагмент микроядра, вычисляющий умножение матриц с использованием ARM SME инструкций.
Fig. 4. Fragment of the microkernel performing matrix multiplication using ARM SVE instructions.

2.2 Интегрированное матричное расширение RISC-V IME

Интегрированное матричное расширение RISC-V IME разрабатывается компанией SpacemiT и по состоянию на конец 2024 года находится в стадии разработки [6]. Это расширение основано на стандартных векторных регистрах векторного расширения RISC-V «V» (RVV) и не добавляет новых регистров управления состоянием и матричных регистров. Матричные регистры образуются путем группировки нескольких векторных регистров в единый блок. Модель программирования с использованием матричных инструкций остается максимально похожей на модель программирования с использованием векторного расширения RVV. Например, одинаковое использование режимов округления, инструкций конфигурации таких, как vsetvli/vsetivli/vsetvl и т. д. В отличие от векторного расширения RVV IME-инструкции выбирают соответствующий матричный блок для умножения матриц и аккумулирования результата на основе значений параметра длины вектора (Vector Length – VL) и размера элемента вектора (Single Element Width – SEW), установленных инструкцией конфигурирования vsetvli/vsetivli/vsetvl. Далее следует небольшой обзор векторного расширения RVV, а после – интегрированного матричного расширения RISC-V IME.

Расширение RVV идейно схоже с расширением ARM SVE. Оно содержит 32 масштабируемых векторных регистра v0-v31, набор векторных инструкций, системные регистры и специальные инструкции для управления динамическим состоянием. Каждый векторный регистр V имеет длину VLEN, неизвестную на этапе компиляции или написании кода и отличающуюся на различных реализациях расширения RVV. Регистры могут быть сгруппированы. Группа регистров может быть использована как единый вектор соответствующего размера. Размер группы может быть динамически изменен и определяется параметром LMUL. Размер элемента в векторе SEW также может быть динамически изменен.

Динамическое состояние процессора, определяющее поведение векторных инструкций, контролируется системными регистрами vector type register(vtype), vector length register(vl), vector byte length register(vlenb), vector start index register(vstart), vector fixed-point rounding mode register(vxrm), vector fixed-point saturation flag(vxsat), vector control and status register(vcsr). Ключевым регистром является регистр vtype. Структура этого регистра изображена на рис. 5.

Конфигурирование системного регистра vtype осуществляется при помощи специальных инструкций vsetvli/vsetivli/vsetvl. Сигнатура инструкций показана на рис. 6.

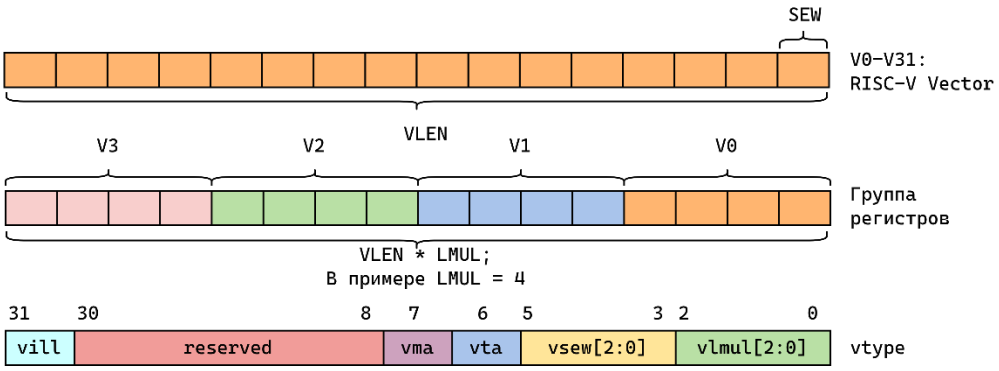


Рис. 5. Векторные регистры расширения RISC-V V; структура системного регистра vtype.
Fig. 5. Vector registers of the RISC-V V extension; structure of the vtype system register;
vill – illegal value if set; vma – vector mask agnostic;.vta – vector tail agnostic;
vsew – selected element width (SEW); vlmul – vector register group multiplier (LMUL).

```
vsetvli rd, rs1, vtypei # rd = new vl, rs1 = AVL, vtypei = new vtype setting
vsetivli rd, uimm, vtypei # rd = new vl, uimm = AVL, vtypei = new vtype setting
vsetvl rd, rs1, rs2 # rd = new vl, rs1 = AVL, rs2 = new vtype value
```

Рис. 6. Сигнатура инструкций конфигурирования системного регистра vtype.
Fig. 6. Signature of instructions configuring the vtype system register.

Инструкции устанавливают значения параметров длины вектора VL, размера элемента вектора SEW и размера группы векторных регистров LMUL. Длина вектора не задается в явном виде. Вместо этого в программе запрашивается необходимая для программы длина вектора (Application Vector Length – AVL). Инструкции vset{i}vl{i} на основе фактической, реализованной аппаратно длины вектора VLEN, а также значений параметров SEW и LMUL вычисляют максимально возможную длину вектора, конфигурируют соответствующим образом регистр vtype и сохраняют вычисленную длину в регистре rd. После этого все векторные инструкции оперируют векторами той длины, которая была сохранена в регистр rd. Регистр rd может быть использован в коде программы, например, для смещения указателей после обработки порции данных. Параметры SEW, LMUL являются операндами инструкций vset{i}vl{i} и могут быть переданы в виде непосредственных констант, закодированных в поле vtypei или через регистр rs2. Значение регистра в этом случае должно содержать эти параметры, закодированные определенным образом.

На рис. 7 приведен пример реализации вычисления saxpy на языке C (Y = a * X + Y) и на ассемблере с использованием инструкций RVV.

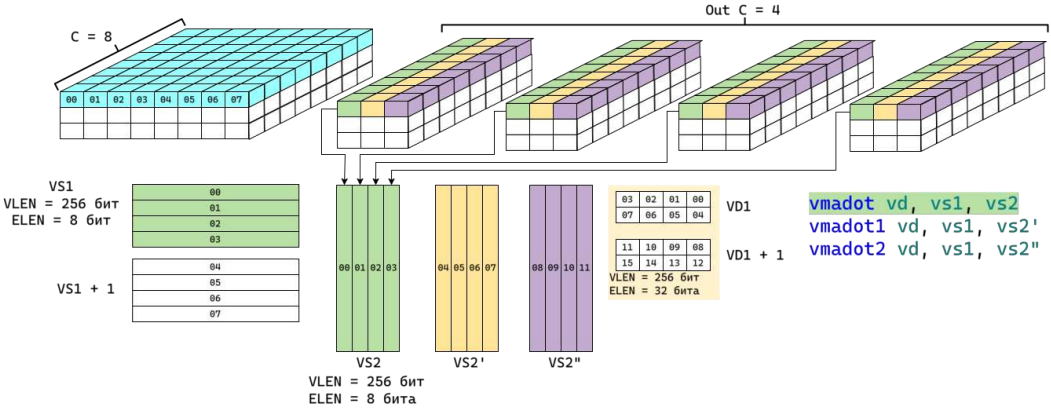
Регистр a0 хранит длину массивов x и y, fa0 – скаляр a, a1 и a2 – адреса начал массивов x и y. В строке 2 выполняется конфигурирование состояния процессора при помощи инструкции vsetvli, в которой указан размер элемента вектора SEW = e32 (элементы по 32 бита), регистры

сгруппированы в векторы по 8 регистров (параметр m8). Доступная длина вектора будет сохранена в регистре a4, а все последующие векторные инструкции будут оперировать векторами с данной длиной. Таким образом, если аппаратная длина VLEN равна 256 битам, то максимальная длина вектора, которая будет сохранена в регистр a4 и записана в VL регистра vtype, будет равна 64 элементам. В строках 3 и 4 происходит загрузка элементов массива x и y в векторы v0 и v8 длиной VL. После этого в строке 5 выполняется инструкция умножения-сложения (a * x + y), результат которой сохраняется в векторном регистре v8. Результат записывается в память в массив y в строке 6. Далее, в строке 7, выполняется уменьшение числа оставшихся для вычисления элементов n на длину вектора VL (которая сохранена в регистре a4). В строке 8 вычисляется длина вектора в байтах. В строках 9-10 смещаются указатели массивов x и y на длину вектора в байтах. В строке 11 выполняется проверка условия выхода из цикла, и управление передается в начало выполнения следующей итерации или на инструкцию выхода из цикла. В случае если остаточная длина массивов n меньше, чем максимальная длина, например 64 элемента, инструкция vsetvli установит параметр VL и значение регистра a4 равным n.

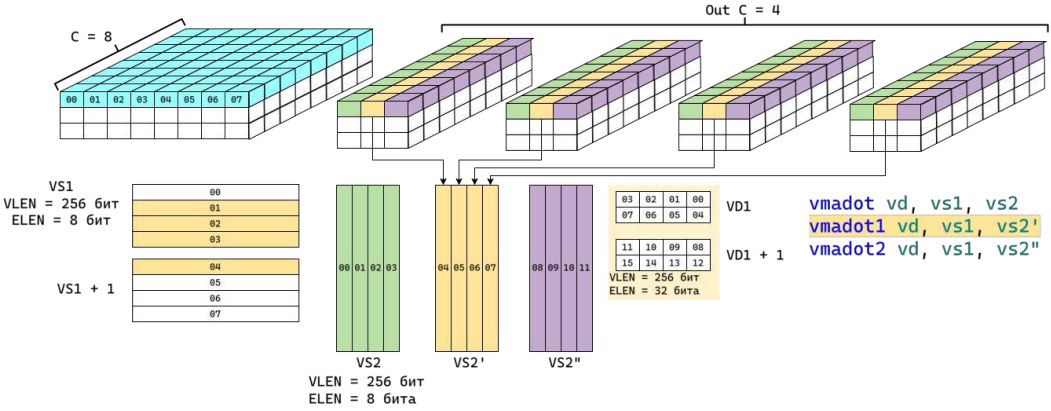
```
# void saxpy(size_t n, const float a,
#            const float *x, float *y) {
#   size_t i;
#   for (i=0; i<n; i++)
#     y[i] = a * x[i] + y[i];
# }
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y
1  saxpy:
2     vsetvli a4, a0, e32, m8, ta, ma
3     vle32.v v0, (a1)
4     vle32.v v8, (a2)
5     vfmacc.vf v8, fa0, v0
6     vse32.v v8, (a2)
7     sub a0, a0, a4
8     slli a4, a4, 2
9     add a1, a1, a4
10    add a2, a2, a4
11    bnez a0, saxpy
12    ret
```

Рис. 7. Пример реализации saxpy на языке C и на ассемблере с использованием RVV инструкций.
Fig. 7. Implementation of saxpy in C and assembly using RVV instructions.

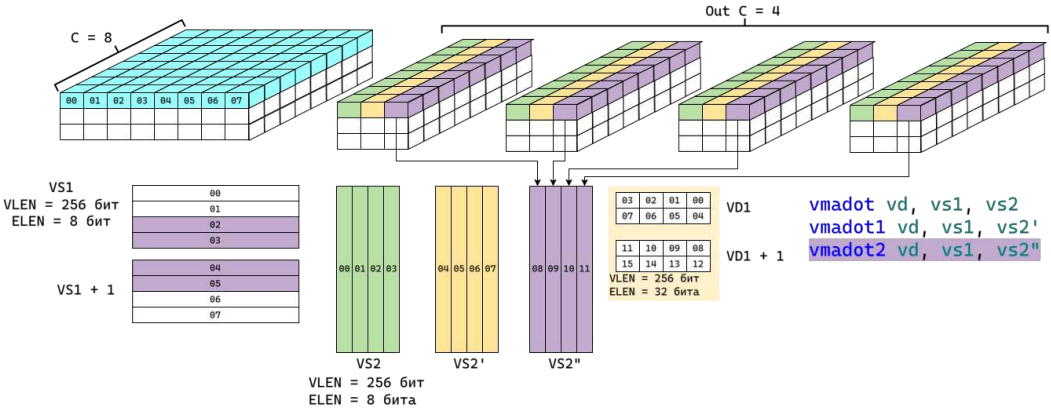
Пример выше демонстрирует основную идею сценария использования RVV, понимание которой необходимо для ознакомления с расширением IME. Расширение RVV представляет одну ключевую инструкцию умножения матриц vmadot (A[M:K] * B[K:N] = C[M:N]), которая может выполняться в двух режимах: в режиме обычного умножения матриц и в режиме «скользящего окна». В режиме «скользящего окна» для умножения из матричного регистра выбирается матрица A со смещением в 1, 2, 3 или n строк. Размеры M, N и K определяются длиной векторных регистров VLEN и выбранным размером элементов вектора SEW [6].



а) выполнение инструкции vmadot vd, vs1, vs2.
a) execution of the vmadot vd, vs1, vs2.



б) выполнение инструкции режима «скользящего окна» vmadot1 vd, vs1, vs2'.
b) execution of the sliding-window vmadot1 vd, vs1, vs2'.



в) выполнение инструкции режима «скользящего окна» vmadot1 vd, vs1, vs2''.
c) execution of the sliding-window vmadot1 vd, vs1, vs2''.

Рис. 8. Пошаговый процесс вычисления фрагмента двумерной свертки с использованием инструкций RISC-V IME; шаг 1 – а; шаг 2 – б; шаг 3 – в.
Fig. 8. Step-by-step process of computing a fragment of a convolution using RISC-V IME instructions; step 1 - a; step 2 - b; step 3 - c.

Режим «скользящего окна» удобен для вычисления операции двумерной свертки методом image2col при помощи операции умножения матриц. Например, если размер входной карты признаков равен 1x3x8x8 (в формате NHWC), а размер ядра свертки – 4x3x3x8 (в формате NHWC), то размер выходной карты признаков будет равен 1x6x6x4 (NHWC). В этом случае применение трех координат каждого из 4 ядер свертки (иначе говоря, одной строки каждого из 4 ядер свертки) может быть вычислено при помощи последовательности из трех инструкций vmadot в режиме «скользящего окна», как показано на рис. 8. Результат выполнения каждой инструкции умножения матриц vmadot будет аккумулироваться в векторном регистре VD.

Расширение RISC-V IME, повторно используя ресурсы регистров векторного расширения, может обеспечить улучшение производительности вычислений искусственных нейронных сетей при очень небольшой стоимости аппаратного обеспечения.

2.3 Матричное расширение RISC-V Attached Matrix Extension

Матричное расширение RISC-V Matrix Multiplication – RVM (Attached Matrix Extension) является независимым матричным расширением процессорной архитектуры RISC-V, которое разрабатывается компанией T-Head (подразделение компании Alibaba). По состоянию на 2024 год расширение RVM находится в разработке. В отличие от интегрированного расширения IME расширение RVM не использует векторные регистры, по этой причине его называют «независимым». Расширение содержит 8 матричных регистров M0-M7, размер которых определяется конкретной реализацией архитектуры. Длина строки каждого матричного регистра составляет RLEN бит, а количество строк равно RLEN / 32. Таким образом, количество столбцов в матричном регистре определяется шириной типа элементов матрицы.

RVM расширение содержит множество инструкций, включающее в себя: инструкции умножения матриц, инструкции загрузки матричных регистров из памяти и сохранения их значений в память, инструкции поэлементных операций над матричными регистрами (поэлементное сложение, умножение, вычитание и др.), инструкции перемещения данных между матричными регистрами, инструкции для обнуления матричных регистров, служебные инструкции конфигурирования размера матричного блока.

2.4 Архитектура DaVinci

Процессорная архитектура DaVinci [7] разработана компанией HiSilicon и ориентирована на решение задач, связанных с выполнением и обучением искусственных нейронных сетей. В 2018 году был анонсирован процессор Ascend AI, реализующий данную архитектуру. Архитектура DaVinci является закрытой, однако в открытых источниках опубликовано некоторое количество работ, позволяющих получить представление о принципах ее организации, а также некоторых особенностях ее функционирования.

На рис. 9 представлена функциональная структура процессорного ядра AI core с архитектурой DaVinci.

Ядро с архитектурой DaVinci содержит три основных вычислительных процессорных элемента: скалярный (Scalar Unit), векторный (Vector Unit) и кубический (Cube Unit) процессорные элементы. Скалярный процессорный элемент выполняет арифметические и логические операции над скалярными переменными, а также выполняет команды, определяющие поток управления (команды условной и безусловной передачи управления). Векторный процессорный элемент выполняет команды, реализующие арифметические и логические операции над векторами данных. Кубический процессорный элемент выполняет команды, реализующие операцию умножения матриц фиксированного размера, и по своей сути является систолическим массивом АЛУ.

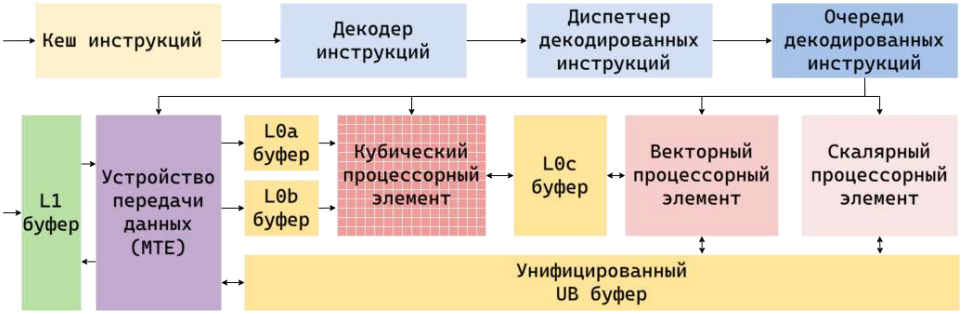


Рис. 9. Функциональная структура архитектуры DaVinci.
Fig. 9. The structure of the DaVinci architecture.

Кроме вычислительных процессорных элементов архитектура DaVinci обладает устройствами передачи данных (Memory Transfer Element – MTE), которые выполняют копирование данных между различными типами памяти. Одновременно с копированием может быть выполнена обработка данных, например преобразование типов из fp32 в fp16, транспонирование копируемого блока данных, вычисление функции активации (например, ReLU) и другие, более сложные, преобразования формата хранения данных.

Архитектура DaVinci реализует неоднородную иерархическую память трех типов: буфер L1, буфер L0 и унифицированный буфер (Unified Buffer – UB). Буфер L0 предназначен для хранения операндов кубического процессорного элемента, то есть умножаемых матриц A и B. Матрица A должна быть размещена в буфере L0a, а матрица B – в L0b. Матрицы должны быть сохранены в блочном формате. Размер блока зависит от типа элементов матриц, например для типа fp16 размер блока составляет 16x16 элементов. Результат умножения матриц сохраняется в буфере L0c также в блочном формате. Унифицированный буфер UB хранит операнды векторных инструкций. Буфер L1 служит для временного размещения часто используемых входных данных, а также для сохранения промежуточных результатов вычислений. Обычный сценарий организации вычислений на ядре с процессорной архитектурой DaVinci включает в себя следующие манипуляции с данными:

1) копирование входных параметров в буфер L1;
2) копирование фрагмента входных данных из буфера L1 в буферы L0a и L0b;
3) вычисление произведения матриц на кубическом процессорном элементе с аккумулированием результата в буфере L0c;
4) копирование результата умножения матриц из буфера L0c в унифицированный буфер UB;
5) обработка данных, скопированных в унифицированный буфер UB, на векторном процессорном элементе;
6) копирование данных из унифицированного буфера UB в буфер L1 для сохранения промежуточных результатов вычислений.

Каждый вычислительный процессорный элемент и устройство передачи данных обладает собственной очередью инструкций. После декодирования инструкций специальное устройство, называемое диспетчером декодированных инструкций, помещает очередную инструкцию в соответствующую очередь. Выполнение инструкций из каждой очереди организовано по принципу FIFO (первым пришел – первым обслужен). Таким образом процессорные устройства и устройства передачи данных могут функционировать одновременно и независимо друг от друга, обеспечивая параллелизм уровня инструкций. Синхронизация функционирования процессорных элементов и устройств передачи данных

осуществляется посредством размещения в соответствующих очередях специальных инструкций. Эти инструкции моделируют обмен сообщениями, содержащими события синхронизации. Иначе говоря, в очередь одного устройства вставляется инструкция отправки события синхронизации, а в очередь другого – инструкция приема этого события. При выполнении инструкции приема устройство блокируется до тех пор, пока не будет получено соответствующее событие синхронизации, т.е. пока другое устройство не выполнит инструкцию отправки.

Кубический процессорный элемент позволяет эффективно выполнять операции умножения матриц. Эффективное выполнение операции двумерной свертки на вычислительном ядре с архитектурой DaVinci возможно путем выражения операции двумерной свертки через операции умножения матриц методом im2col. Так как входные данные для кубического процессорного элемента должны храниться в памяти в блочном формате, то для вычисления двумерной свертки посредством умножения матриц необходимо изменить формат хранения операндов свертки соответствующим образом. Архитектура DaVinci предоставляет специальные инструкции, выполняющие такое преобразование формата многомерных массивов, выполняемое на одном из устройств передачи данных.

Резюмируя выше сказанное, архитектура DaVinci содержит инструкции для выполнения умножения матриц, хранящихся в памяти в блочном формате, а также инструкции преобразования формата хранения многомерных массивов im2col. Ввиду того, что архитектура является закрытой, у авторов данной работы нет возможности продемонстрировать пример реализации умножения матриц с использованием кубических инструкций. Однако, для проектирования абстрактного высокоуровневого представления значимым является лишь наличие в архитектуре матричных инструкций.

2.5 Модель матричного процессора

В результате проведенного анализа матричных расширений и поддерживающих матричные инструкции процессорных архитектур были выявлены следующие основные аспекты:

- наборы матричных инструкций оперируют набором матричных регистров или специальной памятью для хранения их операндов;
- матричные регистры хранят матричные блоки;
- матричные регистры могут использовать векторные регистры или формировать независимый регистровый файл;
- для эффективного использования матричных инструкций требуется блочное хранение матриц;
- матричные процессорные устройства могут не иметь возможности напрямую обращаться к оперативной памяти.

Таким образом, проведенный анализ позволяет предложить следующую модель абстрактной матричной процессорной архитектуры. На рис. 10. представлена общая схема модели.

Модель включает в себя 4 адресных пространства: 1) адресное пространство управляющего устройства; 2) адресное пространство внешнего вычислительного устройства; 3) матричное адресное пространство; 4) векторное адресное пространство. *Адресное пространство управляющего устройства* является абстракцией над оперативной памятью. Предполагается, что данное адресное пространство содержит данные для скалярных и служебных вычислений. *Матричные* и *векторные адресные пространства* служат для хранения операндов матричных и векторных процессорных устройств. *Адресное пространство внешнего вычислительного устройства* является абстракцией над памятью сопроцессора, в случае если матричный процессор реализован в виде отдельного устройства, например память GPU или AI ускорителя (или High Bandwidth Memory – HBM память). Когда матричный процессор реализован в виде специального АЛУ процессорного ядра, это

адресное пространство может отсутствовать или являться абстракцией процессорного кэша высокого уровня.

Предлагаемая модель также содержит следующие абстрактные вычислительные устройства: 1) процессор управляющего устройства; 2) матричное устройство; 3) векторное устройство.

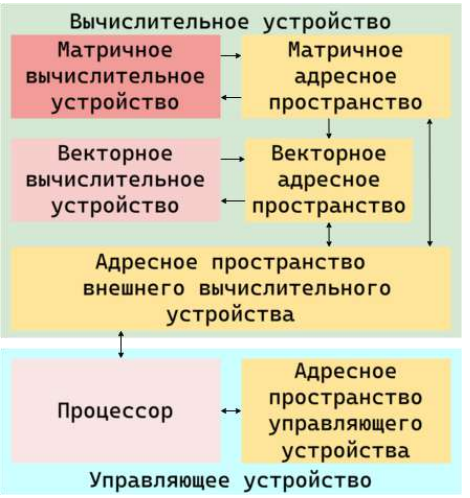


Рис. 10. Модель матричного процессора.
Fig. 10. Model of a matrix processor.

3. Компилятор моделей машинного обучения

Настоящий раздел содержит описание предлагаемого подхода к реализации компиляторов искусственных нейронных сетей для процессорных архитектур, которые поддерживают матричные инструкции. Предлагаемый подход ориентируется на введенную в предыдущем разделе модель абстрактной матричной архитектуры. Для построения компилятора предлагается семейство многоуровневых промежуточных представлений. Реализация представлений может быть выполнена в инфраструктуре MLIR. Далее раскрываются детали предлагаемого семейства промежуточных представлений, а также использование инфраструктуры MLIR для их реализации.

3.1 Обзор инфраструктуры MLIR

Проект многоуровневого промежуточного представления (Multiple Level Intermediate Representation – MLIR) является частью проекта LLVM. Основная цель проекта состоит в предоставлении необходимой единой инфраструктуры для проектирования и разработки промежуточных представлений. Такая инфраструктура, обычно, включает в себя менеджер проходов, движок сопоставления с образцом (поиск соответствия шаблонам) и интерфейсы для трансляции инструкции одних промежуточных представлений в инструкции других и т. д. Тем самым для существующих и вновь создаваемых компиляторов исчезнет необходимость в разработке такой инфраструктуры, а разработчики могут сосредоточить свои усилия на разработке оптимизирующих трансформаций и на проектировании новых промежуточных представлений. MLIR представляет такую инфраструктуру, а также содержит множество встроенных промежуточных представлений.

Инфраструктура MLIR не содержит конкретных инструкций, вместо этого она предоставляет необходимые абстракции для разработки произвольных промежуточных представлений, которые обладают иерархической структурой, например, как у LLVM IR: модуль содержит функции, функции содержат базовые блоки, базовые блоки содержат инструкции.

Представление MLIR состоит из трех ключевых сущностей: 1) операция; 2) регион; 3) блок. Операция – единица, задающая семантику выполнения. Операция может содержать регион. Регион – это контейнер для операций. Регион содержит блоки, которые содержат операции. Блок – это линейная последовательность операций (аналогична базовому блоку). Последняя операция каждого блока содержит операцию передачи управления другому блоку или родительскому региону. Таким образом, представление MLIR обладает иерархической рекурсивной структурой: операция содержит регион, регион содержит блоки, каждый блок содержат операции и т.д.

На рис. 11 представлен пример рекурсивной структуры MLIR-представления.

В этом примере операция builtin.module содержит один регион (в фигурных скобках), в котором неявно содержится один блок. Этот блок включает в себя одну операцию func.func,содержащую один регион, который в свою очередь содержит один явный блок ^BB0. Блок ^BB0 содержит последовательность операций arith.constant и операцию scf.for, которая, в свою очередь, включает в себя регион с одним блоком, содержащем операции memref.load, arith.mulf и т. д. Представление MLIR состоит из трех основных объектов: 1) значения; 2) типы; 3) атрибуты. Значениями являются SSA-имена переменных. В примере на рис. 11 значениями являются %[0-1] и %arg[0-3]. Все значения должны быть типизированными (f32, memref<128xf32>, ...). Атрибуты содержат дополнительную информацию о MLIR-сущностях, например атрибут fastmath = #arith.fastmath<none>.

```
"builtin.module"() ({
  "func.func"() <{
    function_type = (f32, memref<128xf32>, memref<128xf32>) → (),
    sym_name = "saxpy"}> ({
  ^bb0(%arg0: f32, %arg1: memref<128xf32>, %arg2: memref<128xf32>):
    %0 = "arith.constant"() <{value = 0 : index}> : () → index
    %1 = "arith.constant"() <{value = 1 : index}> : () → index
    %2 = "arith.constant"() <{value = 128 : index}> : () → index
    "scf.for"(%0, %2, %1) ({
    ^bb0(%arg3: index):
      %3 = "memref.load"(%arg1, %arg3)
        : (memref<128xf32>, index) → f32
      %4 = "arith.mulf"(%arg0, %3)
        <{fastmath = #arith.fastmath<none>}>
        : (f32, f32) → f32
      %5 = "memref.load"(%arg2, %arg3)
        : (memref<128xf32>, index) → f32
      %6 = "arith.addf"(%4, %5)
        <{fastmath = #arith.fastmath<none>}>
        : (f32, f32) → f32
      "memref.store"(%6, %arg2, %arg3)
        : (f32, memref<128xf32>, index) → ()
      "scf.yield"() : () → ()
    }) : (index, index, index) → ()
    "func.return"() : () → ()
  }) : () → ()
}) : () → ()
```

Рис. 11. Рекурсивная структура MLIR-представления.
Fig. 11. Recursive structure of the MLIR representation.

Инфраструктура MLIR не содержит фиксированного набора инструкций, она лишь предоставляет необходимые инструменты для разработки собственного набора операций, типов и атрибутов. Объединение множества операций, типов и атрибутов формирует диалект. Диалект может быть представлен как библиотека, которая содержит необходимые структуры разработанного промежуточного представления. MLIR содержит множество встроенных диалектов (48 диалектов) для различных сценариев использования: диалект с задающими поток управления операциями; диалект арифметических операций, диалект

аффинных циклов, диалект векторных операций, диалект архитектурно-ориентированных операций и др. Одной из ключевых возможностей инфраструктуры MLIR является определение пользовательских диалектов на специальном предметно-ориентированном языке. Более того, MLIR позволяет описывать в декларативной форме шаблоны для сопоставления с образцом и перезаписи. В следующих разделах содержится описание функциональной структуры компилятора и особенности реализации гетерогенного матричного промежуточного представления.

3.2 Гетерогенное матричное промежуточное представление

Предлагаемая в работе модель матричного процессора содержит процессорные элементы для вычисления матричных и векторных инструкций и несколько типов адресных пространств. Следовательно, гетерогенное матричное промежуточное представление должно содержать необходимые операции для организации вычислений на каждом процессорном элементе, операции копирования данных между различными адресными пространствами, а также служебные операции для диспетчеризации матричных и векторных операций. Служебные операции выполняются вычислительным устройством общего назначения. На нем же выполняются операции копирования данных между адресными пространствами. Так как эффективное использование матричных инструкций требует блочного хранения матриц в памяти, то предлагаемое множество операций содержит соответствующие операции.

В результате был разработан диалект heteroMx, содержащий необходимый набор вычислительных и служебных операций. При описании диалекта используются следующие обозначения. Адресное пространство управляющего устройства (host address space – has) имеет идентификатор, равный 0, и соответствует адресному пространству процесса. Адресное пространство внешнего устройства (device address space – das) имеет идентификатор 1. Матричное адресное пространство (matrix address space – mas) имеет идентификатор 2. Векторное адресное пространство (vector address space – vas) имеет идентификатор 3. Предлагаемый диалект содержит набор операций, представленный в табл. 2.

Реализация операции изменения формата хранения матриц во многом полагается на доступные в целевой архитектуре инструкции. Как правило, удобные для этих целей инструкции содержатся в векторном расширении. Эти инструкции реализуют все различные перестановки элементов в векторных регистрах. В худшем случае эта операция может породить скалярный код, в котором матрица сохраняется в блочном формате поэлементно.

Необходимо отметить, что операции диалекта heteroMx работают как на уровне тензорной семантики, так и на уровне семантики многомерных массивов в памяти. Тензорная семантика позволяет абстрагироваться от адресных пространств конкретного типа и памяти как таковой. Тензоры – это абстрактные N-мерные агрегаты с известным типом элементов и известным рангом. Семантику многомерных массивов в памяти в терминах MLIR задают ссылки на области памяти (memref). Процесс перехода от тензорной семантики к семантике многомерных массивов в памяти называется буферизацией. Инфраструктура MLIR содержит все необходимые примитивы для автоматизации этого процесса, однако часть действий необходимо реализовать на стороне разрабатываемого компилятора, например, выделение буферов под тензоры в адресных пространствах, вычисление размещения буферов в адресных пространствах и др. Эти вопросы не будут раскрыты в данной работе, а являются темой будущих исследований. Описание диалекта, а также его операций выполняется в декларативной форме.

Предлагаемый подход к построению компилятора использует доступные в инфраструктуре MLIR диалекты. К числу таких диалектов относятся диалекты крупноблочных операций нейронных сетей, диалект для организации аффинных гнезд циклов, диалект emitc и др. В следующем разделе представлено описание общей структуры компилятора.

Табл. 2. Операции диалекта heteroMx.
Table 2. Operations of the heteroMx dialect.

heteroMx.mad a:Tensor<*>, b:Tensor<*>, c:Tensor<*>	Операция умножения матриц c = a × b
heteroMx.blocking src:Tensor<*>, dst:Tensor<*>	Операция, преобразующая формат хранения матриц в блочный
heteroMx.unblocking src:Tensor<*>, dst:Tensor<*>	Операция, восстанавливающая исходный формат хранения матриц
heteroMx.element_wise_(add/sub/mul/div/sqrt/max/min/and/or/rsqrt/relu/...) src1:Tensor<*>, src2:Tensor<*>, dst:Tensor<*>	Поэлементные векторные операции
heteroMx.reduce_(sum/prod/min/max) src:Tensor<*>, dst:Tensor<*>	Операция вычисления свертки вектора
heteroMx.zero dst:Tensor<*>	Операция очистки операнда. Операнд может быть размещен во всех доступных адресных пространствах
heteroMx.copy_has_to_das src:Tensor<*>, dst:Tensor<*> heteroMx.copy_das_to_has * heteroMx.copy_das_to_mas * heteroMx.copy_das_to_vas * heteroMx.copy_mas_to_das * heteroMx.copy_vas_to_das * heteroMx.copy_vas_to_mas * heteroMx.copy_mas_to_vas *	Операции копирования соответствующих буферов памяти
heteroMx.kernel_run @func_name	Операция запуска функции, которая является вычислительным ядром на матричном процессоре

3.3 Функциональная структура компилятора

Компилятор моделей машинного обучения (AI-компилятор) в качестве входной программы принимает описание вычислительного графа модели на DSL-языке в одном из возможных форматов, например ONNX, PyTorch, TensorFlow и др. На данном этапе могут быть применены преобразования модели, специфичные для конкретной библиотеки машинного обучения. Вычислительный граф модели должен быть преобразован в программу на соответствующем диалекте. Для большинства популярных библиотек машинного обучения существуют соответствующие диалекты. Так проект onnx-mlir реализует onnx диалект, torch-mlir – torch диалект и т.д. Также существуют трансляторы из исходного кода DSL-языка в операции соответствующего диалекта.

Следующим шагом необходимо транслировать модель в высокоуровневый диалект, независимый от конкретной библиотеки машинного обучения. Диалект на данном уровне содержит крупноблочные операции, характерные для искусственных нейронных сетей:

операции двумерной свертки, умножения матриц и т.д. Существуют сторонние проекты, которые реализуют диалекты с крупноблочными операциями, например stablehlo, mhlo, chlo диалекты. Однако инфраструктура MLIR содержит стандартизированный набор крупноблочных тензорных операций в TOSA [8] диалекте (Tensor Operator Set Architecture – TOSA). Предлагаемый AI-компилятор использует данный диалект. На рис. 12 изображена функциональная структура AI-компилятора.

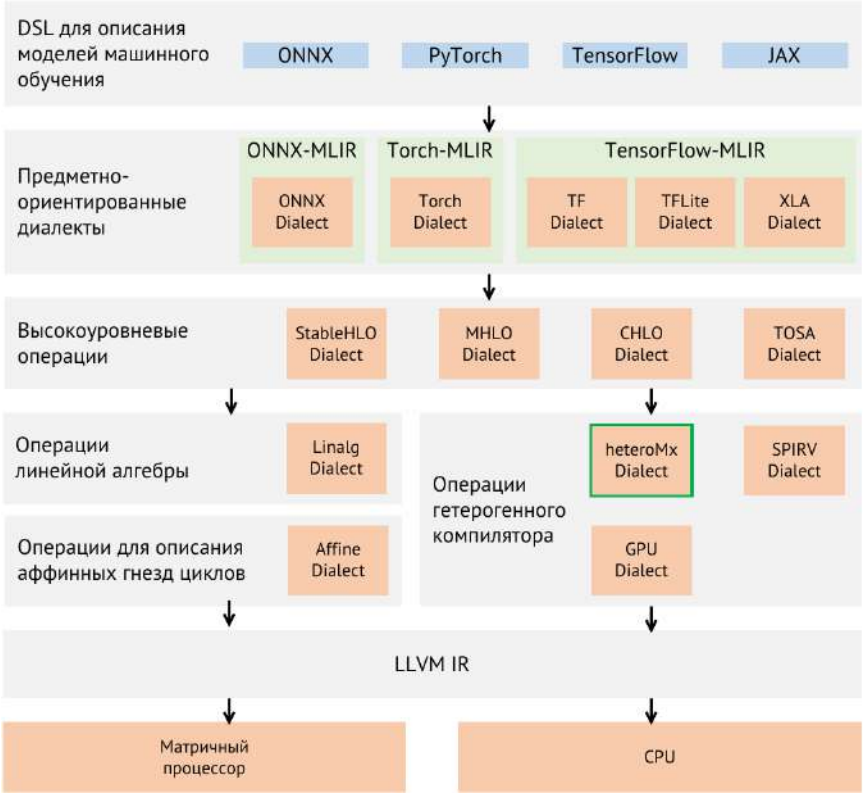


Рис. 12. Функциональная структура AI-компилятора.
Fig. 12. The structure of the AI compiler.

На уровне TOSA диалекта могут выполняться простые оптимизации для сокращения избыточных операций, например избыточные транспонирования вида transpose(transpose(A) = A. MLIR позволяет описать подобные преобразования в декларативной форме на специальном языке. На рис. 13 продемонстрирован пример описания такого преобразования.

```
/// transpose(transpose(in)) → in
def TransposeElimination : Pat<
  (Tosa_TransposeOp (Tosa_TransposeOp $in, $perms_in), $perms_out),
  (replaceWithValue $in),
  [(Constraint<CPred<"$0 == $1">> $perms_in, $perms_out)]>;
```

Рис. 13. Описание шаблона для устранения избыточных операций транспонирования.
Fig. 13. Description of the pattern for eliminating redundant transpose operations.

Следующим шагом TOSA-программа транслируется в linalg диалект, который содержит операции линейной алгебры. Часть операций, такие, как умножение матриц и двумерные свертки, транслируются в операции разработанного диалекта heteroMx. После этого

программа транслируется в диалект affine, и в ней появляются аффинные циклы. На уровне операций диалекта affine, а также в процессе трансляции в операции разработанного диалекта heteroMx, выполняется разбиение циклов на блоки циклов (tiling циклов), а также матриц (в терминах работы Goto и Geijn [9] – упаковка блоков матриц). Наличие явных аффинных циклов позволяет применять полиэдральные преобразования циклов.

После этого программа, представленная в диалектах affine и heteroMx, поэтапно, прогрессивно (то есть с малым шагом интерпретации) транслируется в LLVM IR и далее в код целевой архитектуры матричного процессора. На данном этапе могут быть использованы архитектурно-ориентированные диалекты, например, arm_sme для использования инструкций матричного расширения архитектуры ARM или SPIRV для генерации вычислительных ядер гетерогенной архитектуры или другие диалекты. Однако, этот этап трансляции находится за рамками данной работы и, несомненно, является важнейшим, с большим количеством открытых научных вопросов.

Результатом компиляции, при таком подходе, может быть LLVM код (совместно с SPIRV кодом) или машинный код, в случае если трансляция шла по вышеобозначенному пути. На этапе, когда программа представлена в диалектах affine и heteroMx, возможен альтернативный путь. Программа может быть транслирована в emitc диалект, и тогда результирующим артефактом будет программа на языке C++, которая может быть скомпилирована в код целевой архитектуры любым доступным компилятором.

4. Заключение

Развитие матричных расширений процессорных архитектур, а также внедрение этих расширений в специализированные AI-процессоры, позволяет существенно повысить эффективность выполнения искусственных нейронных сетей. В работе выполнен обзор базовых функциональных возможностей некоторых популярных матричных расширений процессорных архитектур, в частности расширений ARM SME, RISC-V IME, RISC-V AME, а также процессорной архитектуры DaVinci. В результате анализа нами была предложена модель абстрактного матричного процессора, которая отражает особенности современных процессорных архитектур, поддерживающих матричные расширения.

Для введенной модели матричного процессора нами разработано гетерогенное матричное промежуточное представление. Предложенное промежуточное представление может быть использовано для построения компиляторов нейронных сетей. Работа затрагивает только промежуточные представления высокого уровня, представления среднего и низкого уровня остаются в качестве направлений для дальнейших исследований. Предложенное промежуточное представление нами было реализовано в инфраструктуре MLIR в виде диалекта heteroMx. В работе описан подход к построению AI-компилятора с использованием разработанного диалекта heteroMx. Гетерогенное матричное промежуточное представление может быть адаптировано или конкретизировано для других матричных процессорных архитектур, которые не были рассмотрены в настоящей работе, но имеют сходства с предложенной моделью.

Предложенное промежуточное представление и разработанный диалект могут быть использованы при реализации гетерогенного компилятора, например, в качестве высокоуровневого преставления в компиляторе, реализующем стандарт SYCL для матричной целевой архитектуры.

Список литературы / References

[1]. Sousa R. et al. Tensor slicing and optimization for multicore NPUs //Journal of Parallel and Distributed Computing. – 2023. – Т. 175. – С. 66-79.

[2]. Jouppi N. P. et al. In-datacenter performance analysis of a tensor processing unit //Proceedings of the 44th annual international symposium on computer architecture. – 2017. – С. 1-12.

[3]. Lattner C. et al. MLIR: Scaling compiler infrastructure for domain specific computation //2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2021. – С. 2-14.
[4]. Remke S., Breuer A. Hello SME! Generating Fast Matrix Multiplication Kernels Using the Scalable Matrix Extension //arXiv preprint arXiv:2409.18779. – 2024.
[5]. Stephens N. et al. The ARM scalable vector extension //IEEE micro. – 2017. – Т. 37. – №. 2. – С. 26-39.
[6]. The RISC-V IME Set Specification. https://github.com/space-mit/riscv-ime-extension-spec/releases/download/v0429/spacemit-ime-asciidoc.pdf.
[7]. H. Liao, J. Tu, J. Xia and X. Zhou, "DaVinci: A Scalable Architecture for Neural Network Computing," 2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, 2019, pp. 1-44, doi: 10.1109/HOTCHIPS.2019.8875654.
[8]. TOSA specification. https://www.mlplatform.org/tosa/tosa_spec.html, Accessed July 2023.
[9]. Goto K., Geijn R. A. Anatomy of high-performance matrix multiplication //ACM Transactions on Mathematical Software (TOMS). – 2008. – Т. 34. – №. 3. – С. 1-25.

Информация об авторах / Information about authors

Иван Иванович КУЛАГИН – кандидат технических наук, научный сотрудник ИСП РАН. Область научных интересов: построение компиляторов, оптимизирующие компиляторы, полиэдральная компиляция, генерация кода, модели параллельного программирования, AI-ускорители.

Ivan Ivanovich KULAGIN – Cand. Sci. (Tech.), Researcher in ISP RAS. Research interests: compiler construction, compiler optimizations, polyhedral compilation, code generation, parallel programming models, AI-accelerators.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Михаил Вячеславович ПАНТИЛИМОНОВ – научный сотрудник отдела компиляторных технологий. Научные интересы: статический анализ, компиляторные технологии, СУБД.

Mikhail Vyacheslavovich PANTILIMONOV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, DBMS.

Андрей Викторович ВЯЗОВЦЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Andrey Viktorovich VYAZOVTSEV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Михаил Максимович РОМАНОВ – студент ВМК МГУ, лаборант отдела компиляторных технологий ИСП РАН. Сфера научных интересов: компиляторные технологии, ускорение искусственных нейронных сетей.

Mikhail Maksimovich ROMANOV – a student of CMC department of MSU, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: compiler technologies, artificial neural networks acceleration.

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий Института системного программирования с 2004 года. Сфера научных интересов: компиляторные оптимизации, динамическая (JIT) компиляция.

Dmitry Mikhailovich MELNIK – Senior Researcher in Compiler Technology department. Research interests: compiler optimizations, dynamic (JIT) compilation.

DOI: 10.15514/ISPRAS-2025-37(1)-6

CC BY

Фреймворк автоматизации тестирования на гонки по данным

Е.А. Герлиц, ORCID: 0000-0002-1747-075X <gerlits@ispras.ru>
В.С. Мутилин, ORCID: 0000-0003-3097-8512 <mutilin@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В данной работе рассматривается класс параллельных программ над общей памятью и присущий этому классу программ тип ошибок – гонки по данным. Нами спроектирован тестовый фреймворк для разработки сценариев тестирования на гонки по данным по аналогии с широко применяемыми тестовыми фреймворками для последовательных программ. Основной проблемой при создании теста является недетерминизм, присущий выполнению многопоточных программ. С целью обеспечить повторяемость сценария тестирования в данной работе мы вводим понятие точек синхронизации в исходном коде программы, в которых тест регулирует порядок исполнения инструкций программы потоками при помощи внедрения синхронизационных действий. Разработанные тесты на гонки по данным выполняются полностью автоматически при помощи фреймворка и могут использоваться для регрессионного тестирования.

Ключевые слова: гонка по данным; состояние гонки; воспроизведение гонки по данным; тестирование на гонки по данным; многопоточная программа; параллельная программа с общей памятью; тестовый фреймворк; фреймворк автоматизации тестирования; синхронизация потоков.

Для цитирования: Герлиц Е.А., Мутилин В.С. Фреймворк автоматизации тестирования на гонки по данным. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 107–120. DOI: 10.15514/ISPRAS–2025–37(1)–6.

Towards a Test Automation Framework for Data Race Testing

E. Gerlits ORCID: 0000-0002-1747-075X <*gerlits@ispras.ru*>
V. Mutilin ORCID: 0000-0003-3097-8512 <*mutilin@ispras.ru*>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper, we examine shared memory concurrent programs and errors occurring in them, specifically data races. We design a test automation framework to develop data race revealing testing scenarios by analogy to test automation frameworks for sequential programs. The main problem complicating development of testing scenarios is the nondeterministic nature of multithreaded program executions. To provide repeatable testing scenarios we define a notion of synchronization points in the source code of the computer program where a test regulates execution of parallel threads with synchronization actions. Tests being developed with our test automation framework can be executed automatically and can be used for regression testing.

Keywords: data race; race condition; data race reproducing; data race testing; multithreaded program; shared memory concurrent program; testing framework; test automation framework; thread synchronization.

For citation: Gerlits E.A., Mutilin V.S. Towards a test automation framework for data race testing. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 107-120 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-6.

1. Введение

В выполнении многопоточной программы присутствует гонка по данным, когда два потока обращаются к общей памяти без синхронизации, причём одно из обращений – это запись. Гонка по данным может приводить к некорректным вычислениям в программе и исключительным ситуациям. Поэтому задача проверки многопоточных программ на гонки по данным является актуальной.

Одним из возможных способов проверки является тестирование при помощи сценариев тестирования, программируемых вручную. Если проводить аналогию с тестированием последовательных программ, то этот способ тестирования реализуется при помощи тестовых фреймворков [1-3]. Однако если сценарии тестирования для последовательных программ создаются на основе функциональных требований к программе или её компонентам, то проверки многопоточных программ на гонки по данным на этапе функционального тестирования сравнительно редки на практике. Тем не менее они востребованы в иных случаях:

- В процессе циклической отладки [4-6] гонки по данным фактически создаётся тест, который воспроизводит выполнение программы с гонкой по данным – тест на реальную ошибку.
- При помощи этого теста проверяют отсутствие гонки по данным после исправления ошибки в исходном коде.
- Этот же тест может далее запускаться в ходе регрессионного тестирования, чтобы препятствовать проникновению гонки по данным в будущие версии программы.
- Код теста документирует основной результат отладки гонки по данным, которая может быть весьма трудоёмкой и длительной.

Таким образом, задача ручной разработки сценариев тестирования для выявления гонок по данным в многопоточных программах является актуальной.

Для автоматизации выполнения множества таких тестовых сценариев и получения результатов тестирования можно применять существующие тестовые фреймворки для последовательных программ. Однако их оказывается недостаточно:

- Многопоточным программам свойственен недетерминизм выполнения из-за непостоянства скорости исполнения инструкций ядрами процессора в многозадачной среде и других причин. В результате в повторном выполнении программы порядок исполнения потоками инструкций чтения и записи в память меняется. Как следствие, переменные получают иные значения, ход выполнения программы меняется и гонка по данным не возникает в повторном выполнении программы. Недетерминизм выполнения многопоточных программ мешает обеспечить основное свойство сценариев тестирования – повторяемость.
- Для осуществления проверок в тестах и вывода результатов этих проверок фреймворк предоставляет специальные функции – assert functions. Однако гонки по данным могут проявляться в виде ошибок или исключительных ситуаций не во всех выполнениях программы, в которых они происходят. Например, операция добавления элемента в список и операция проверки списка на пустоту, выполненные без синхронизации из разных потоков, не приводят к ошибке, если список изначально был не пуст. Разработка сценария тестирования, на котором гонка по данным проявляется в виде ошибки (обнаруживается при помощи assert functions), может быть намного более трудоёмкой задачей, чем разработка сценария тестирования, на котором гонка по данным обнаруживается более эффективными динамическими методами [7-10].

В данной работе мы предложим средства для решения обозначенных проблем и объединим эти средства с функциональностью существующих тестовых фреймворков, тем самым обеспечив возможность создания автоматически выполняемых тестов на гонки по данным по аналогии с тестами для последовательных программ. Разработанный нами фреймворк можно применять как в процессе циклической отладки гонки по данным для пошаговой разработки теста, воспроизводящего гонку по данным, так и для регрессионного тестирования на гонки по данным в системах непрерывной интеграции и развёртывания.

Работа устроена следующим образом. В разделе 2 мы приводим компонентную структуру фреймворка и последовательно её поясняем. Раздел 3 посвящён автоматизации выполнения тестов. В разделе 4 мы обсуждаем проблемы, которые возникли на практике в ходе разработки тестов при помощи фреймворка. Раздел 5 посвящён обзору связанных работ. Выводы по результатам исследования сделаны в разделе 6.

2. Фреймворк автоматизации тестирования на гонки по данным

На рис. 1 компонентная структура фреймворка для тестирования последовательных программ расширяется дополнительными компонентами (они выделены серым цветом), которые необходимы для построения тестов, нацеленных на гонки по данным.

2.1 Среда выполнения тестового фреймворка для последовательных программ

При наличии нескольких тестов на гонки по данным их последовательный запуск предлагается организовывать при помощи одного из существующих фреймворков для тестирования последовательных программ [1-3]. На рис. 1 показан случай, когда тесты компилируются вместе с кодом программы в один исполняемый образ. В этом случае за автоматизацию запуска тестов отвечает часть фреймворка для последовательных программ, которую мы называем средой выполнения фреймворка и которая также компилируется вместе с программой в один исполняемый образ. Выполнение каждого теста в свою очередь является последовательностью из трёх шагов:

1. *setup* – подготовка состояния программы для теста.

2. *test case* – многопоточный тестовый сценарий, в котором функции тестируемой программы вызываются из нескольких потоков, взаимодействующих через общую память.
3. *tear down* – перевод программы в начальное состояние для запуска следующего теста.

Рис. 1. Компонентная структура фреймворка для автоматизации выполнения тестов на гонки по данным.

Fig 1. Component structure of a framework automating execution of tests for data races.

2.2 Happens-before детектор

Тестовые фреймворки для последовательных программ предоставляют тестам функции для осуществления проверок и вывода (сохранения) результатов этих проверок – assert functions. При помощи этих функций гонки по данным можно обнаруживать по их проявлениям – ошибочным вычислениям или исключительным ситуациям.

В данной работе предлагается применять альтернативный способ обнаружения гонок по данным, а именно использовать один из существующих инструментов [4, 11] динамического мониторинга отношения happens-before [12] на множестве исполнений инструкций программы. Такой инструмент обнаруживает не последствия гонок по данным, а исполнения пар инструкций доступа к памяти без синхронизации, в которых один из доступов к памяти является записью и выполняется неатомарно.

Применение happens-before детектора вместо обнаружения гонки по данным по её проявлению облегчает разработку ручных сценариев тестирования так как:

- Количество выполнений программы, в которых два доступа к памяти, образующих гонку по данным, выполняются без синхронизации, не меньше количества выполнений, в которых гонка по данным проявляется в виде ошибки.
- Гонка по данным возникает до её проявления в виде наблюдаемой ошибки. Чтобы гонка по данным проявилась, требуется дополнительно организовать выполнение программы по определённому сценарию от момента её возникновения до момента её проявления.

Существуют альтернативные инструменты динамического обнаружения гонок по данным, которые могут быть более эффективны для конкретного класса программ. Так гонки по данным в операционной системе Linux ищут при помощи инструмента KCSAN [10].

2.3 Расстановка вызовов wait/notify функций

У недетерминированного выполнения многопоточной программы может быть несколько источников: непостоянная скорость исполнения инструкций программы ядрами процессора в многозадачной среде, вытесняющий планировщик потоков в операционной системе, использование случайных чисел, недетерминированное поведение внешних систем и системных вызовов и др. В данной работе ограничимся учётом только одного основного и непременного источника недетерминизма, которым является непостоянная скорость исполнения инструкций ядрами процессора.

Следствием непостоянной скорости исполнения инструкций ядрами процессора является изменение порядка исполнения инструкций программы потоками. Функции *wait* и *notify* используются для регулирования этого порядка. Основная идея состоит в следующем. Если требуется, чтобы некоторое исполнение инструкции i_1 в одном потоке произошло после некоторого исполнения инструкции i_2 в другом потоке, то в исходный код программы перед инструкцией i_1 вставляется вызов функции *wait,* которая приостанавливает выполнение текущего потока до уведомления, а после инструкции i_2 вызов функции *notify*, которая это уведомление отправляет.

Одна и та же инструкция программы может исполняться несколько раз разными потоками. Реализации функций *wait* и *notify* должны уметь определять целевые исполнения инструкций i_1 и i_2 соответственно, на которых они должны срабатывать. Принимать решение функции могут на основе аргументов, переданных им в качестве параметров, и внутреннего состояния (собственных глобальных переменных). Аргументами, в частности, могут быть значения глобальных и локальных переменных самой программы, доступные в местах вставки вызовов функций.

Каждый тест реализует некоторый набор *wait/notify* функций, вызовы которых вставляются в исходный код программы вручную. Так как согласно рис. 1 тесты компилируются вместе с программой в один исполняемый образ, то вызовы *wait/notify* функций необходимо добавить в исходный код программы сразу для всех тестов. Однако во время выполнения конкретного теста должны исполняться *wait* и *notify* функции только этого теста. Для этого положим, что тесты пронумерованы. Тогда при вставке вызова функции *wait* или *notify* в исходный код можно обрамлять вызов функции условным оператором в виде *if(test==n){wait/notify(…)}*, где *n* – номер теста, к которому относится этот вызов. Определение глобальной целочисленной переменной *test* должно быть частью среды выполнения фреймворка. Значение этой переменной устанавливает сам тест в начале своего выполнения в функции *setup*.

Основной проблемой является выявление мест в исходном коде, в которые следует добавить вызовы *wait/notify* функций, чтобы гонка по данным воспроизвелась. К сожалению, ручной способ расстановки *wait/notify* функций, как и всякий нетривиальный способ отладки, трудно алгоритмизировать и формально обосновать. На практике мы следовали следующему подходу:

1. Идентифицируем пару исполнений инструкций обращения к памяти, которые участвуют в гонке по данным. Будем называть эти исполнения инструкций целевыми. Если создаётся тест на существующую гонку по данным, ранее обнаруженную инструментом динамического анализа, то такой инструмент должен выдавать некоторую информацию о паре целевых исполнений инструкций (адреса инструкций, стеки вызовов функций во время их исполнения и др.).
2. Подобрать расстановку вызовов *wait/notify* функций, которая стабильно воспроизводит целевые исполнения инструкций.
3. Если happens-before детектор не сообщает о гонке по данным между целевыми исполнениями инструкций, то устранить отношение порядка между ними,

скорректировав расстановку *wait/notify* функций.

4. Убедиться, что гонка по данным воспроизводится стабильно. Для этого можно воспользоваться предложенным нами фреймворком, чтобы организовать многократные запуски теста с автоматической проверкой обнаружения целевой гонки по данным в каждом запуске.

3 Автоматизация добавления вызовов wait/notify функций в исходный код программы

Функции *wait/notify* выполняются только во время тестирования, поэтому обращения к ним, расставленные в исходном коде программы вручную, являются вспомогательным кодом. С увеличением количества тестов исходный код программы загромождается вспомогательным кодом. Это негативно сказывается на сопровождении исходного кода программы. Также некоторые стандарты безопасной разработки, такие как DO-178C [13], не допускают наличие исходного кода, который никогда не выполняется в реальных (не тестовых) запусках.

Автоматическое инструментирование исходного кода программы вызовами *wait/notify* функций позволяет устранить эту проблему. Предлагается следующий подход:

- В каждом тесте в некотором виде задаётся отношение между вызовами *wait/notify* функций (буквально текстовыми строками, содержащими вызов *wait/notify* функции со списком фактических параметров) и местами в исходном коде, в которые их нужно вставить.
- Компонент фреймворка проходит по всем тестам, получает от них вышеобозначенные отношения и вставляет вызовы *wait/notify* функций в указанные места в исходном коде.

***Определение 1**. Точка синхронизации (ТС) – это место в исходном коде программы для вставки вызовов wait/notify функций.*

Точка синхронизации может быть указана по-разному. В простейшем случае в качестве идентификатора точки синхронизации (ИТС) можно взять пару *(p, l),* где $p: \mathbb{N} \to \Theta$ – путь к модулю исходного кода в файловой системе, $l \in \mathbb{N}$ – номер строки, Θ – некоторое множество символов – алфавит, из которого составляются строки, например, пути в файловой системе, исходный код программы и др.

Исходный код программы может изменяться во время разработки и сопровождения, а идентификатор в виде пары *(p, l)* не устойчив ко многим этим изменениям. Чтобы повысить устойчивость, мы задаём точки синхронизации вручную в исходном коде в виде комментариев специального вида */*prefix n*/*, где *prefix* – некоторая строка символов, одинаковая для всех комментариев специального вида и позволяющая отделить их от других комментариев, а *n* – номер точки синхронизации, и нумеруем точки синхронизации локально в пределах функции, метода, класса.

***Определение 2**. Идентификатор точки синхронизации (ИТС) – это четвёрка (p, c, f, n), где p, c, f:* $\mathbb{N} \to \Theta$, $n \in \mathbb{N}$, Θ *– алфавит:*

- *p – путь в файловой системе к модулю исходного кода;*
- *c – имя класса;*
- *f – сигнатура функции или её часть, например, имя функции;*
- *n – порядковый номер точки синхронизации.*

Если *c* и *f* не заданы, то точка синхронизации установлена для модуля исходного кода. Иначе если *f* не задана, то точка синхронизации установлена для класса. Иначе если *c* не задан, то точка синхронизации установлена для функции вне класса. Иначе точка синхронизации установлена для метода.

Указанный метод идентификации точек синхронизации удобен для совместной работы, потому что локальная нумерация точек синхронизации позволяет без конфликтов параллельно нумеровать различные сущности (различные функции, методы, классы, структуры и файлы).

Отметим, задаваемый таким образом идентификатор точки синхронизации может оказаться не уникальным. Например, в случае вложенных (nested) одноимённых классов или функций, расположенных в одном модуле исходного кода. Но такие случаи редки и могут быть легко устранены – мы просто ведём общую нумерацию у одноимённых сущностей в пределах одного модуля исходного кода.

Определение 3. *Синхронизационное действие (СД) – это исходный код вызова функций wait или notify с указанием фактических параметров.*

Указание фактических параметров означает, что синхронизационное действие предполагает его вставку в конкретную точку синхронизации. Соответствующее отношение {ТС} → {СД} между множеством точек синхронизации в программе и множеством синхронизационных действий теста задаётся в тесте вручную путём перечисления пар (*ИТС, СД)* в отдельном текстовом файле.

Компонентная структура фреймворка изображена на рис. 2. Она расширяет компонентную структуру фреймворка из рис. 1 автоматическим инструментированием программы синхронизационными действиями тестов.

Рис. 2. Компонентная структура фреймворка для тестирования на гонки по данным с автоматическим инструментированием исходного кода вызовами wait/notify функций.

Fig 2. Component structure of a framework for data race testing which includes automatic source code instrumentation with wait/notify function calls.

3.1 Алгоритм инструментирования исходного кода программы синхронизационными действиями тестов

Псевдокод алгоритма представлен на листинге 1. В алгоритме используются следующие обозначения:

- *S* – множество всевозможных строк $\mathbb{N} \to \Theta$, где Θ – множество символов (алфавит).
- *ID* – множество всевозможных идентификаторов точек синхронизации.

Алгоритм предполагает, что некоторый скрипт предварительно прошёл по каталогу с тестами и в директории каждого теста прочитал файл с отношением {ТС} → {СД}, интерпретировал его и сформировал для алгоритма аргумент *sync* – отношение между номерами тестов и функциями, отображающими идентификаторы точек синхронизации на синхронизационные

действия теста. Аналогично сформирован аргумент *header* – отношение между номерами тестов и заголовочными файлами (путями к ним в файловой системе), в которых объявлены *wait/notify* функции, используемые тестом. Аргумент *src* моделирует содержимое инструментируемых файлов исходного кода программы в виде функции, которая путям к файлам в файловой системе ставит в соответствие функцию, отображающую номера строк этих файлов на исходный код в этих строках.

Вход: $header : \mathbb{N} \to S$ {номер теста -> путь к заголовочному файлу теста}
$sync : \mathbb{N} \to (ID \to S)$ {номер теста -> (ИТС -> СД)}
$src : S \to (\mathbb{N} \to S)$ {путь к файлу кода -> (номер строки -> код)}
Выход: src - инструментированный код

$a : \mathbb{N} \to S, a = \varnothing \to \mathbb{N}$ {номер СД -> код СД}
$t : \mathbb{N} \to \mathbb{N}, t = \varnothing \to \mathbb{N}$ {номер СД -> номер теста}
$i : ID \to \mathbb{N}, i = \varnothing \to \mathbb{N}$ {ИТС -> номер СД}
$f : S \to ID, f = \varnothing \to ID$ {путь к файлу кода программы -> ИТС}
for all $tid \in Dom(sync)$ **do**
  for all $sid \in Dom(sync(tid))$ **do**
    $n = |Dom(a)| + 1$
    $a = a \cup \{(n, sync(tid)(sid))\}$
    $t = t \cup \{(n, tid)\}$
    $i = i \cup \{(sid, n)\}$
    $f = f \cup \{(sid.f, sid)\}$
for all $fpath \in Dom(f)$ **do**
  $fsid = \{sid \in Im(f) : (fpath, sid) \in f\}$
  $l : ID \to \mathbb{N}, l = SidToLine(fsid, fpath)$ {ИТС -> номер строки}
  $m = 0$
  $h = \varnothing$
  while $|Dom(l)| > 0$ **do**
    $(sid, line) \in l : \forall l_2 \in Im(l) : line <= l_2$ {Выбор ТС по порядку}
    $l = l \backslash (sid, line)$
    for all $n \in Im(i) : (sid, n) \in i$ **do**
      $code = Concat(\text{“}if(test ==\text{”}, Itoa(t(n)), \text{“})\{\text{”}, a(n), \text{“}\}\backslash n\text{”})$
      $src(fpath) = Insert(src(fpath), line + m, code)$
      $h = h \cup \{hpath : (t(n), hpath) \in header\}$
      $m = m + 1$
  for all $hpath \in h$ **do**
    $code = Concat(\text{“}\#include <\text{”}, hpath, \text{“}> \backslash n\text{”})$
    $src(fpath) = Insert(src(fpath), 1, code)$

Листинг 1. Алгоритм инструментирования исходного кода программы синхронизационными действиями тестов.
Listing 1. Algorithm for program source code instrumentation with the test synchronization actions.

Алгоритм инструментирования вызывает ряд тривиальных вспомогательных функций, которые мы не будем приводить:

- *Concat*: $(\mathbb{N} \to \Theta) \times (\mathbb{N} \to \Theta) \longrightarrow (\mathbb{N} \to \Theta)$ – конкатенация последовательностей (строк).
- *Itoa*: $\mathbb{N} \longrightarrow (\mathbb{N} \to \{\text{´0´}, ..., \text{´9´}\})$ – перевод числа в строку.
- *Insert*: $(\mathbb{N} \to \Theta) \times \mathbb{N} \times \Theta \longrightarrow (\mathbb{N} \to \Theta)$ – вставка элемента из множества Θ в последовательность $\mathbb{N} \to \Theta$ перед элементом, позиция которого задаётся целым из множества $\mathbb{N}$.

Функция *SidToLine* выполняет поиск номеров строк, в которых располагаются точки синхронизации. На вход функция принимает множество *fsid* идентификаторов точек синхронизации, которые находятся в одном модуле исходного кода программы. Для идентификаторов вида $(p, l) : p \in S, l \in \mathbb{N}$, где p – путь к файлу, а l – номер строки, алгоритм функции *SidToLine(fsid, fpath)* представлен на листинге 2. Точный алгоритм поиска номеров строк для идентификаторов точек синхронизации по определению 2 предполагает синтаксический анализ модуля исходного кода, в котором находятся точки синхронизации. Краткое описание такого алгоритма приведено в практической части работы в разделе 4.3.

```
function SidToLine(fsid, fpath)
  if ∃sid ∈ fsid then
    {(sid, sid.l)} ∪ SidToLine(fsid\{sid}, fpath)
  else
    ∅
```

Листинг 2. Псевдокод функции SidToLine
Listing 2. Pseudocode of SidToLine function

4 Результаты практического применения фреймворка

Разработанный нами фреймворк, в частности, применялся в проекте по созданию открытой распределённой операционной системы для умных устройств. При помощи фреймворка было разработано несколько регрессионных тестов на гонки по данным, обнаруженные инструментом TSan. Реализация алгоритма инструментирования исходного кода синхронизационными действиями тестов позволила запускать указанные тесты автоматически на любой версии кода программы по изменению (commit) в системе непрерывной интеграции и развёртывания.

По результатам практического применения фреймворка выявлено две проблемы:

1. Наведение частичного порядка между исполнениями инструкций программы из-за синхронизации внутри *wait/notify* функций.
2. Взаимная блокировка потоков внутри *wait* функций.

Возникла также необходимость идентификации точек синхронизации по определению 2, так как указанный способ идентификации точек синхронизации устойчив к некоторым изменениям исходного кода: добавление новых сущностей, удаление существующих сущностей, изменение и перемещение сущностей, не содержащих точки синхронизации, и др. Эта устойчивость позволяет запускать тесты на множестве версий программы с относительно небольшими изменениями, как это происходит в случае автоматических запусков тестов в системах непрерывной интеграции и развёртывания.

4.1 Наведение частичного порядка между инструкциями программы

Ожидание *wait* одним потоком уведомления *notify* от другого предполагает взаимодействие через общую память. Чтобы это взаимодействие не приводило к гонкам по данным, в реализациях *wait/notify* функций необходимо применять средства синхронизации потоков или атомарные операции. Однако ввиду транзитивности отношения happens-before, синхронизация потоков, инициированная *wait/notify* функциями, наводит порядок между исполнениями инструкций самой программы. Чтобы этого не происходило можно помечать исходный код (*wait/notify* функции) специализированными аннотациями, которые предоставляет happens-before детектор, чтобы исключить этот код из наблюдения. Для языка C++ также можно использовать атомарные операции с указанием порядка std::memory_order_relaxed [14].

4.2 Взаимная блокировка потоков

Данная проблема особенно актуальна, если автоматический тест на гонку по данным используется для регрессионного тестирования. В течение жизненного цикла программы её код исправляется и дорабатывается. В результате может возникнуть версия программы, инструментирование которой синхронизационными действиями теста может привести к взаимной блокировке потоков, то есть к ситуации, когда два потока выполняют функции *wait*, бесконечно ожидая уведомление друг от друга. Чтобы эту ситуацию обнаружить, можно либо установить глобальный таймаут для теста, либо выполнять ограниченное по времени ожидание в функциях *wait* с уведомлением о таймауте.

4.3 Поиск номеров строк для точек синхронизации

Получить номера строк для идентификаторов точек синхронизации по определению 2 можно следующим способом. Перед инструментированием выполнить вспомогательную сборку программы с автоматическим перехватом опций компилирования файлов. Сохранить опции компилирования в хранилище (базе данных или файле). Для этого можно применить, например, инструмент Clade [15]. Реализовать функцию *SidToLine*, которая:

1 Извлекает опции компилирования файла *fpath* из хранилища.

2 Использует эти опции для трансляции файла *fpath* в абстрактное синтаксическое дерево (АСД), которое можно анализировать программно. Для исходного кода на языке C++ эта задача может быть решена, к примеру, при помощи библиотеки libclang [16].

3 Для каждого идентификатора *sid* из *fsid*:

3.1 По идентификатору *sid* находит в АСД объект комментарий *obj*.

3.2 От объекта *obj* получает номер строки *end*, в которой *obj* завершается.

3.3 Добавляет пару *(sid, end+1)* к отношению, возвращаемому функцией.

5. Обзор связанных работ

Если фреймворк, предложенный в настоящей работе, нацелен на разработку тестовых сценариев вручную, то фреймворк, предложенный в работе [17], нацелен на автоматический отбор подмножества тестов для регрессионного тестирования. Фреймворк выявляет общие переменные программы, то есть переменные, к которым выполняется доступ из нескольких потоков программы. Для этих целей применяется статический анализ escape analysis [18-19]. Далее фреймворк определяет подмножество общих переменных, которые затронуты изменениями в программе. На следующем шаге отбирается подмножество тестов, которые ищут гонки по данным между обращениями к этим переменным. На последнем шаге фреймворк упорядочивает тесты по убыванию количества общих переменных, обращения к которым проверяются на гонки по данным.

В качестве альтернативы ручной разработке сценария тестирования на конкретную гонку по данным можно рассматривать метод автоматического поиска (подбора) сценария тестирования, который эту гонку по данным воспроизводит (обнаруживает). Так в работе [20] инструмент CHESS отслеживает события в программе, такие как обращения к сервисам операционной системы, которые используются для организации многопоточных вычислений, и на основе наблюдений строит модель выполнения программы в виде happens-before графа [12]. Эта модель предполагает последовательную консистентность вычислений [21]. Из модели вычисления инструмент извлекает информацию о потенциально возможных альтернативных последовательностях переключений потоков – расписаниях. Выполнение программы по выбранному расписанию организуется при помощи управления планировщиком потоков. Параллельно с регулированием выполнения программы

инструмент CHESS отслеживает возникающие ошибки, в частности, гонки по данным. Если при выполнении программы по некоторому расписанию обнаружена ошибка, то инструмент помечает (сохраняет) это расписание, чтобы выполнение с ошибкой можно было повторить с целью отладки.

Ручная разработка сценариев тестирования на гонки по данным подразумевает итеративный подход, когда в процессе разработки тест запускается, чтобы на основе анализа (отладки) его выполнения понять, как тест должен быть доработан, например, поправлена расстановка *wait/notify* функций. Итеративному сценарию мешает недетерминизм выполнения многопоточных программ, из-за которого два последовательных запуска одной и той же версии теста приводят к существенно различным выполнениям программы. Чтобы эту проблему преодолеть, можно применять методы записи и воспроизведения выполнения программ. Так метод Instant Replay, изложенный в работе [4], моделирует все способы межпоточного взаимодействия как операции над разделяемыми объектами в общей памяти. На этапе мониторинга метод Instant Replay сохраняет информацию о событиях чтения и записи в разделяемые объекты, а на этапе воспроизведения на основе этой информации организует повторное выполнение программы таким образом, чтобы на каждом шаге все потоки читали те же данные, которые они читали на этапе мониторинга.

В нашем фреймворке для обнаружения гонок по данным используется инструмент динамического анализа – happens-before детектор. Метод happens-before не требует, чтобы гонка по данным реально происходила, то есть чтобы два доступа к памяти выполнялись одновременно либо один за другим. Метод happens-before обнаруживает, что два доступа к памяти выполняются без синхронизации (не упорядочены), один из них выполняет запись и один из них выполняется неатомарно. При этом, если инструмент не наблюдает какие-то события синхронизации, то он может выдавать ложные предупреждения о гонках по данным. Чтобы убедиться в реальности гонки по данным, можно продолжить уточнять сценарий тестирования до тех пор, пока гонка по данным фактически не произойдёт. В работе [22] решается схожая задача. На вход анализу подаётся трасса выполнения программы. В этой трассе при помощи метода lock-set [8] выявляется гонка по данным, то есть в трассе обнаруживаются два обращения к одному участку памяти из разных потоков, одно из них выполняет запись и пересечение множеств захваченных блокировок в момент обращений к памяти пусто. Далее по трассе и гонке по данным автоматически выполняется поиск возможного порядка исполнения инструкций программы потоками, на котором два обращения к памяти, участвующие в гонке по данным, происходят одно за другим, то есть воспроизводится гонка по данным. Этот поиск выполняется статически за счёт формулирования задачи поиска в виде формулы для SMT (Satisfiability modulo theories) [23] решателя [24].

В индустрии разработки программного обеспечения существуют инструменты, автоматизирующие функциональное тестирования программ. Некоторые подобные инструменты поддерживают разработку тестов для обнаружения состояний гонки в многопоточных программах. Примером является инструмент RapiTest [25]. Инструмент предлагает табличный формат для описания тестовых сценариев, а для регулирования порядка выполнения потоков предлагает вставлять задержки при помощи оператора wait (ожидание). Ошибки обнаруживаются при помощи традиционных проверок результатов вычислений.

6. Выводы

Разработка тестовых сценариев для обнаружения или воспроизведения гонок по данным зачастую представляет собой трудную задачу. Основной проблемой является недетерминизм, присущий выполнению многопоточных программ. По аналогии с тестированием

последовательных программ, чтобы облегчить разработку тестовых сценариев на гонки по данным, в работе предлагается применять тестовый фреймворк.

Описана компонентная структура такого фреймворка. Основные отличия от фреймворков для последовательных программ – это использование динамического happens-before анализа для обнаружения гонок по данным и регулирование порядка исполнения инструкций программы при помощи добавления в исходный код программы вызовов функций в парадигме wait/notify (ожидать/уведомить).

Разработанные при помощи фреймворка тесты на гонки по данным выполняются автоматически, поэтому могут применяться в системах непрерывной интеграции и развёртывания для регрессионного тестирования.

Список литературы / References

[1]. Beust C., Suleiman H. Next generation Java testing: TestNG and advanced concepts. – Pearson Education, 2007.

[2]. Massol V. JUnit in action. – 2004.

[3]. Hyuk Myeong. Googletest In Practice: Unit Testing Guide for C++ Programmers. – 2021.

[4]. Leblanc. Debugging parallel programs with instant replay //IEEE Transactions on Computers. – 1987. – Т. 100. – №. 4. – С. 471-482.

[5]. Wang Y. et al. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing //Proceedings of annual IEEE/ACM international symposium on code generation and optimization. – 2014. – С. 98-108.

[6]. Thane H., Hansson H. Using deterministic replay for debugging of distributed real-time systems //Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000. – IEEE, 2000. – С. 265-272.

[7]. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice //Proceedings of the workshop on binary instrumentation and applications. – 2009. – С. 62-71.

[8]. Savage S. et al. Eraser: A dynamic data race detector for multithreaded programs //ACM Transactions on Computer Systems (TOCS). – 1997. – Т. 15. – №. 4. – С. 391-411.

[9]. Erickson J. et al. Effective {Data-Race} Detection for the Kernel //9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10). – 2010.

[10]. The Kernel Concurrency Sanitizer. The Linux Kernel Organization. Available at: https://docs.kernel.org/dev-tools/kcsan.html, accessed 04.01.2025.

[11]. Adve S. V. et al. Detecting data races on weak memory systems //ACM SIGARCH Computer Architecture News. – 1991. – Т. 19. – №. 3. – С. 234-243.

[12]. Lamport L. Time, clocks, and the ordering of events in a distributed system //Concurrency: the Works of Leslie Lamport. – 2019. – С. 179-196.

[13]. RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification.

[14]. ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++.

[15]. Исходный код инструмента Clade для перехвата команд сборки. Веб-ссылка: https://github.com/17451k/clade, доступно 01.04.2024.

[16]. Документация компилятора Clang с главой, посвящённой библиотеке libclang. Веб-ссылка: https://clang.llvm.org/doxygen/index.html, доступно 01.04.2024.

[17]. Yu T., Srisa-an W., Rothermel G. SimRT: An automated framework to support regression testing for data races //Proceedings of the 36th international conference on software engineering. – 2014. – С. 48-59.

[18]. Halpert R. L., Pickett C. J. F., Verbrugge C. Component-based lock allocation //16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007). – IEEE, 2007. – С. 353 - 364.

[19]. Huang J., Liu P., Zhang C. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs //Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. – 2010. – С. 207-216.

[20]. Musuvathi M. et al. Finding and Reproducing Heisenbugs in Concurrent Programs //OSDI. – 2008. – Т. 8. – №. 2008.

[21]. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs //IEEE transactions on computers. – 1979. – Т. 100. – №. 9. – С. 690-691.

[22]. Said, M., Wang, C., Yang, Z., & Sakallah, K. (2011, April). Generating data race witnesses by an SMT-based analysis. In NASA Formal Methods Symposium (pp. 313-327). Springer, Berlin, Heidelberg.
[23]. De Moura L., Bjørner N. Satisfiability modulo theories: introduction and applications //Communications of the ACM. – 2011. – Т. 54. – №. 9. – С. 69-77.
[24]. Dutertre B., De Moura L. A fast linear-arithmetic solver for DPLL (T) //International Conference on Computer Aided Verification. – Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. – С. 81-94.
[25]. https://www.rapitasystems.com/products/rapitest.

Информация об авторах / Information about authors

Евгений Анатольевич ГЕРЛИЦ – научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: методы контроля и обеспечения качества программного обеспечения, методы динамической верификации, статической верификации и анализа программ, формальные методы.

Evgeny Anatolievich GERLITS – researcher at the Software Engineering Department of ISP RAS. Main research interests: software quality control and assurance, dynamic and static software verification and analysis, formal methods.

Вадим Сергеевич МУТИЛИН – старший научный сотрудник Института системного программирования. Сфера научных интересов: статический и динамический анализ программ.

Vadim Sergeevich MUTILIN – senior researcher at the Software Engineering Department of the Institute for System Programming of the RAS. Main research interests: static and dynamic program analysis.

DOI: 10.15514/ISPRAS-2025-37(1)-7

CC BY

Методика поиска уязвимостей в программном обеспечении, написанном на нескольких языках программирования

[1, 2, 3] *Б.А.Позин, ORCID: 0000-0002-0012-2230 <bpozin@ec-leasing.ru>*
[2] П.А. *Бородушкина, ORCID: 0009-0009-8897-9025 <paborodushkina@edu.hse.ru>*
[2] *Д.А. Коротков, ORCID: 0009-0004-3064-8980 <dakorotkov_1@edu.hse.ru>*
[2] *М.А. Федоров, ORCID: 0009-0005-5473-9868 <mafedorov_6@edu.hse.ru>*
[2] *А.Ф. Муратов, ORCID: 0009-0009-5044-616X <afmuratov@edu.hse.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Научно-исследовательский университет Высшая Школа Экономики Россия, 101978, Мясницкая ул., д.20.*

[3] *ЗАО ЕС-лизинг, Россия, 117405, Москва, Варшавское ш. д.125.*

Аннотация. В связи с необходимостью повышения производительности труда при анализе (разметке) результатов автоматизированного поиска уязвимостей в программах, проведенного с использованием инструментального средства, возникает проблема дефицита на рынке высококвалифицированных аналитиков для проведения разметки результатов. В работе описана разработанная методика для поиска уязвимостей в программном обеспечение (Далее - ПО), написанном на нескольких языках программирования (C, C++, Java, Python, Go). В ходе ее разработки проведен анализ всех автоматически обнаруживаемых детекторов в программах на этих языках и элементов их конструкций, подлежащих анализу. Детекторы упорядочены в соответствии с классификацией регулятора. Применение методики позволяет снизить квалификационные требования к аналитикам, проводящим разметку, и подготовить таких специалистов в разрабатывающих компаниях.

Ключевые слова: уязвимость; статический анализ программ; детектор; разметка; квалификационные требования.

Для цитирования: Позин Б.А., Бородушкина П.А., Коротков Д.А., Федоров М.А., Муратов А.Ф. Методика поиска уязвимостей в ПО, написанном на нескольких языках программирования. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 121–132. DOI: 10.15514/ISPRAS–2025–37(1)–7.

Благодарности: Авторы признательны д.ф.-м.н. А.А. Белеванцеву за плодотворное обсуждение работы.

Vulnerability Detection Methodology in Software Written in Several Programming Languages

[1,2,3] *B.A. Pozin, ORCID: 0000-0002-0012-2230 <bpozin@ec-leasing.ru>*
[2] *P.A. Borodushkina, ORCID: 0009-0009-8897-9025 <polinaborod21@gmail.com>*
[2] *D.A. Korotkov, ORCID: 0009-0004-3064-8980 < dakorotkov_1@edu.hse.ru >*
[2] *M.A. Fedorov, ORCID: 0009-0005-5473-9868 <mafedorov_6@edu.hse.ru>*
[2] *A.F. Muratov, ORCID: 0009-0009-5044-616X <afmuratov@eu.hse.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *HSE University, 20, Myasnitskaya Ulitsa, Moscow, 101978, Russia*

[3] *«EC-leasing» Co. 125 Varshavskoye shosse, Moscow, 117405, Russia*

Abstract. Due to the need to increase labor productivity when analyzing (marking up) the results of an automated vulnerability search in programs conducted using SAST (Static Application Security Testing) tool, there is a problem of a shortage in the market of highly qualified analysts to mark up the results. The paper describes a developed technique for finding vulnerabilities in software written in several programming languages (C, C++, Java, Python, Go). During its development, an analysis of all automatically detectable detectors in programs in these languages and elements of their structures to be analyzed was carried out. The detectors are ordered according to the classification of the regulator. The application of the methodology allows reducing the qualification requirements for analysts conducting markup and training such specialists in developing companies.

Keywords: vulnerability; static analysis of programs; detector; markup; qualification requirements.

For citation: Pozin B.A., Borodushkina P.A., Korotkov D.A., Fedorov M.A., Muratov A.F. Vulnerability detection methodology in software written in several programming languages. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 121-132 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)- 7.

Acknowledgements. Authors are grateful to Dr. A.A. Belevantsev for a fruitful discussion of the work.

1. Введение

В течение последних нескольких лет на рынке стали доступными инструментальные средства статического анализа программ для поиска уязвимостей и недекларированных возможностей. По данным [1] эти средства используются более, чем в 40% проектов. Повышение эффективности статического анализа программ с целью поиска уязвимостей и недекларированных возможностей в этой связи является чрезвычайно актуальным. Издан стандарт ГОСТ Р 71207-2024 [2]. В этом стандарте предусмотрено проведение разметки (анализа) результатов автоматизированного статического анализа с целью формирования начальной базы предупреждений о потенциальных ошибках и последующего обнаружения истинных и ложноположительных результатов [3-5]. В то же время существует проблема дефицита на рынке высококвалифицированных аналитиков для проведения разметки результатов статического анализа [6]. Данная работа посвящена разработке методики проведения разметки программ, написанных на языках C/C++, Java, Python, Go, которая позволит обучать специалистов для ее проведения и соответственно снижать требования к их начальной квалификации, а также трудоемкость и стоимость проведения таких работ.

2. Постановка задачи

С развитием информационных технологий, возникла необходимость в обеспечении безопасности программ от уязвимостей и недекларированных возможностей, начиная с самых ранних этапов разработки и на всём жизненном цикле ПО [2]. Появился запрос на проведение статического анализа программ, при котором осуществляют проверку кода программы на наличие уязвимостей и недекларированных возможностей. Требования к автоматизации статического анализа и последующей валидации его результатов (разметке) установлены в стандарте [2].

В результате проверки программы статическим анализатором им формируется база данных предупреждений об уязвимостях в той или иной программе.

Содержание этой базы доступно для анализа путем отображения средством доступа к базе. Так, в случае использования статического анализатора Svace [4], средством отображения является понятный для пользователя интерфейс Svacer. Svacer позволяет аналитику оставлять заметки в базе и сравнивать результаты разных запусков анализатора.

Результаты запуска статического анализатора для повышения качества рекомендовано подвергать дополнительному анализу. Этот процесс анализа называют разметкой результата прогона статического анализатора, или просто *разметкой*.

Целью разметки является рассмотрение экспертом предупреждений на предмет истинности и выявления ошибок первого рода (ложноположительные) и ошибок второго рода (ложноотрицательные). Из-за особенностей работы статических анализаторов, определенная часть предупреждений могут быть ложноположительными (данное понятие подразумевает под собой то, что анализатор указывает на конкретный участок кода с предупреждением о возможной уязвимости, однако данный участок кода является безопасным). Ложноотрицательные предупреждения – те, которые не обнаружены анализатором, но известно, что ошибочная конструкция в программе присутствует и может реализоваться при работе программы.

Задача состоит в том, чтобы:

- описать процесс работы по выявлению уязвимости программы на языке программирования, приводящий к устранению максимального количества остаточных уязвимостей после выполнения нескольких последовательных запусков анализатора после устранения найденных на предыдущем запуске уязвимостей;
- провести работу по разметке результата запуска статического анализатора с учетом свойств конкретного языка, на котором написана программа, с достижением высоких показателей по обнаружению и классификации выявленных уязвимостей.

Собственно, разметка и проводится для соотнесения найденной статическим анализатором потенциальной уязвимости (далее называемой - детектором) с реальной уязвимостью, соответствующей одному из выбранных регулятором типов – по классификации [12].

Разрабатываемый процесс должен быть унифицирован для большинства языков программирования и должен применяться любыми аналитиками по разметке в составе разрабатывающей организации.

3. Решение

Для такого рода проблемы потребовалось создание решения, которое позволит за достаточно небольшой период времени подготовить на рынке значительное количество экспертов по разметке за счет сокращения длительности их обучения и снижения квалификационного порога для их включения в проекты в данной сфере. Это позволит за относительно короткий период разрабатывающим компаниям создать в своей структуре команды, умеющие выявлять уязвимости в программах, написанных на различных языках программирования.

3.1 Процесс работы по систематическому выявлению уязвимостей

Процесс работы по систематическому выявлению уязвимостей представлен на *Рис. 1*.

В цикле работ по анализу результатов запуска статического анализатора с учетом свойств конкретного языка осуществляется проверка наличия выявленных Svace срабатываний, или потенциальных уязвимостей: детекторов (детектор - конкретный тип уязвимости и степень ее критичности [13]) и маркеров уязвимостей (предупреждение Svace о позиции уязвимости в исходном файле). При их обнаружении последовательно проводят анализ предполагаемых уязвимостей в соответствии с методикой, описанной ниже, выявление и фиксацию срабатываний, интерпретацию истинных срабатываний в терминах классификации ФСТЭК [12] и анализ выявленных уязвимостей. Далее отчет об обнаруженных уязвимостях передают на устранение. После доработки файл вновь передают на анализ статическим анализатором. Если после автоматизированного анализа маркеров уязвимости найти не удалось, процесс завершается. Если маркеры есть, то цикл повторяется.

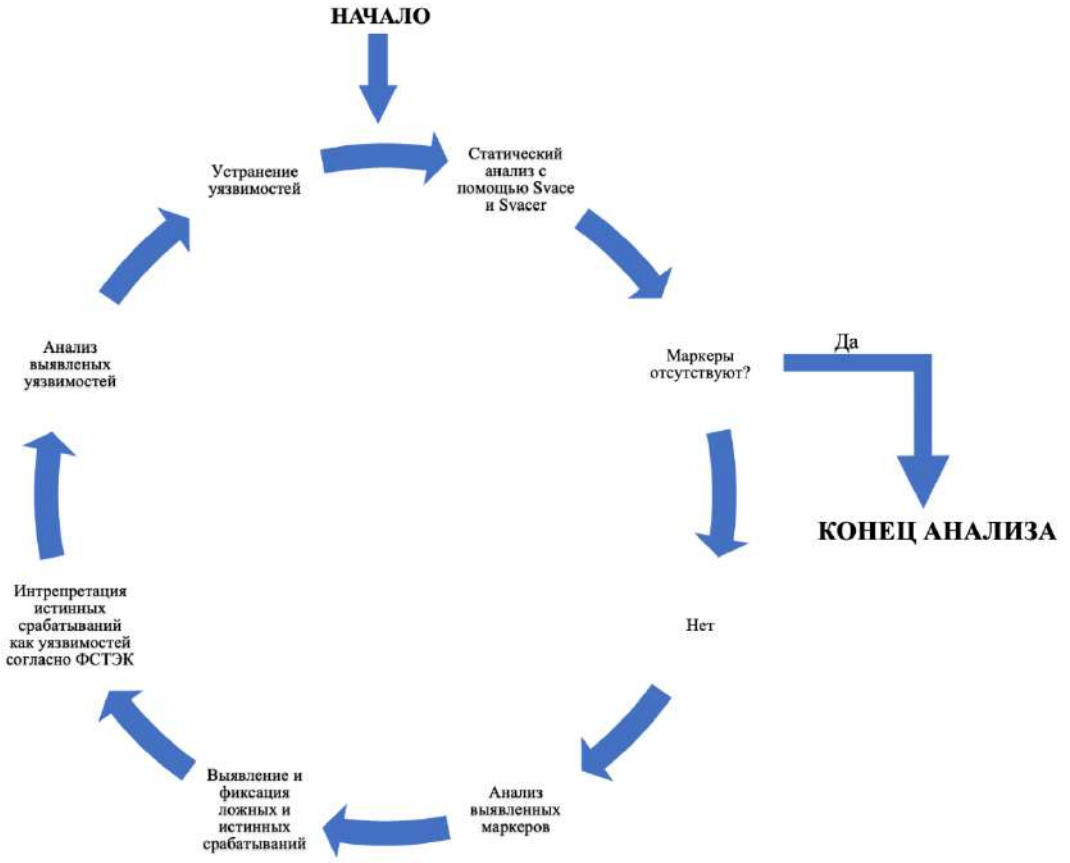


Рис. 1. Схема статического анализа.
Fig. 1. Static Analysis Scheme.

Для работы с результатами анализа используется Svacer, который представляет собой понятный для пользователя интерфейс, обеспечивающий возможность просмотра каталога найденных дефектов, фиксацию результатов верификации аналитиком найденных дефектов как уязвимостей и сравнение результатов анализа после разных запусков Svace.

Следует отметить, что Svace обнаруживает пять степеней критичности уязвимостей: Critical, Major, Normal, Minor, Undefined. С учетом того, что по функциональности и видам анализа

Svace полностью соответствует требованиям стандарта [2], процент истинных срабатываний составляет 60-90% [5]. Проведение разметки позволяет практически достигнуть этих показателей для анализируемых программ.

3.2 Базовые положения для методики разметки программ

Методика разметки программ, разработанная в данной работе, основывается на исследовании потенциально уязвимых конструкций языка программирования, которые обнаруживаются в программах. Для пяти языков программирования исследованы все типы конструкций, обнаруживаемых анализатором Svace как детекторы. Функциональные возможности Svace в этой части приведены в тТабл. 1.

Табл. 1 Функциональные возможности Svace.
Table 1. Svace functionality.

Язык программирования	Всего уязвимостей	Уровни критичности				
		Critical	Major	Normal	Minor	Undefined
C/C++	589	82	165	92	184	66
Go	118	20	30	35	18	15
Java	752	40	301	83	318	10
Python	42	4	7	0	7	24
Всего	150	146	503	210	527	115

В ходе работ над методикой для каждого языка программирования описаны все типы уязвимостей применительно ко всем реализующим их конструкциям каждого языка программирования. Фактически по каждому из пяти языков программирования разработаны справочники по оценке конструкций, в которых описаны уязвимости (детекторы), и правила анализа этих уязвимостей (см. раздел 4). Это описание включено в состав документа «Методика статического анализа для программ на языке». Методика включена в состав автоматизированной системы Центр кибербезопасности [14].

При разработке методики выявления уязвимостей и недекларированных возможностей в программном обеспечении для каждого из языков с применением Svace проведена идентификация всех возможных детекторов как потенциальных уязвимостей согласно классификации, предложенной регулятором [12]. Пример такой идентификации приведен в Табл. 2:

- **Первая группа**: Функции и процедуры, относящиеся к разным прикладным программам и несовместимые между собой (не функционирующие в одной операционной среде) из-за конфликтов, связанных с распределением ресурсов системы.
- **Вторая группа**: Функции, процедуры, изменение определенным образом параметров которых позволяет использовать их для проникновения в операционную среду информационной системы персональных данных (Далее – ИСПДн) и вызова штатных функций операционной системы, выполнения несанкционированного доступа без обнаружения таких изменений операционной системой.
- **Третья группа**: Фрагменты кода программ ("дыры", "люки"), введенные разработчиком, позволяющие обходить процедуры идентификации, аутентификации, проверки целостности и др., предусмотренные в операционной системе.
- **Четвертая группа**: Отсутствие необходимых средств защиты (аутентификации,

проверки целостности, проверки форматов сообщений, блокирования несанкционированно модифицированных функций и т.п.).

- **Пятая группа**: Ошибки в программах (в объявлении переменных, функций и процедур, в кодах программ), которые при определенных условиях приводят к сбоям, в том числе к сбоям функционирования средств и систем защиты информации, к возможности несанкционированного доступа к информации.

Табл. 2 Пример распределения детекторов в Python по типам уязвимостей согласно [12].
Table 2. An example of the distribution of detectors in Python by vulnerability type according to [12].

	Первая группа	Вторая группа	Третья группа	Четвертая группа	Пятая группа
Critical				tainted_array_index tainted_int tainted_ptr tainted_ptr.format_string	
Major		wrong_arguments_order		division_by_zero.ex catch.no_body	invariant_result similar_branches return_in_finally bad_copy_paste
Normal					
Minor	user.malloc_zero_light_demo			division_by_zero.ex.float user.bad_string_demo user.bad_keyword_demo user.bad_string_light_demo user.bad_keyword_light_demo	passed_to_proc_after_release
Un-defined			hardcoded_password	mypy.syntax mypy.var_annotated	accidental_tuple mutable_default_argument mypy.misc mypy.return mypy.operator mypy.attr_defined mypy.valid_type mypy.index mypy.call_overload mypy.arg_type mypy.union_attr mypy.return_value mypy.assignment mypy.list_item mypy.call_arg mypy.override mypy.no_redef mypy.name_defined mypy.str_format mypy.has_type mypy.duplicate_module

3.3 Методика разметки программ

В процессе разметки осуществляется автоматизированный поиск детекторов методами статического анализа, выявление ложноположительных срабатываний в результате работы аналитика по разметке, а также анализ выявленных уязвимостей (правильных срабатываний) на предмет выработки рекомендаций по устранению.

В результате применения методики аналитик по разметке по существу должен ответить на вопрос, являются ли выявленные при статическом анализе детекторы уязвимостями и насколько сильно они сказываются на безопасности написанной программы? Для проведения указанных работ аналитику по разметке необходим доступ (с использованием Svacer) к базе результатов статического анализа и документ «Методика статического анализа для программ на языке».

Основные шаги методики таковы:

1. После получения результатов от Svace и открытия их в веб интерфейсе Svacer приступить к работе. При анализе следует учитывать, что для каждого языка количество детекторов может разниться для разных уровней критичности ПО по сравнению с табл. 1. От аналитика по разметке требуется каждый выявленный детектор соотносить с имеющимися в методике и проверять в соответствии с контекстом программы на наличие ложноположительных выявленных уязвимостей.
2. Для того, чтобы выявить истинные и ложные срабатывания, необходимо:
   - рассмотреть контекст, то есть понять назначение кода и входные данные, которые он обрабатывает;
   - проверить код вручную: проверить наличие признаков ложных срабатываний, таких как использование безопасных методов программирования, правильность обработки ошибок или наличие комментариев, объясняющих отсутствующую уязвимость;
   - оценить критичность детектора;
   - для выявленной уязвимости выработать рекомендации по ее устранению.
3. Аналитику по разметке рекомендуется в ходе работ или по их завершении обсудить результаты анализа с коллегами (аналитиками), разработчиками конструкций, оцененных как уязвимость, а также со специалистами по безопасности, чтобы получить независимое «второе» мнение, прежде всего по поводу влияния выявленных уязвимостей или сомнительных конструкций на безопасность системы, в состав которой включено рассматриваемое ПО. В результате таких консультаций могут быть получены дополнительные рекомендации по написанию тестов, специально предназначенных для проверки отсутствия уязвимости. Могут быть получены и рекомендации по необходимости применить другие инструменты SAST или динамического анализа программ (DAST) для подтверждения результатов. В частности, может быть получена рекомендация использовать инструменты для отслеживания потока данных, чтобы проверить, может ли ошибочная конструкция действительно привести к нарушению безопасности (в неочевидных случаях).
4. Документировать результаты разметки в виде отчета, который направляется в соответствии с технологией работ организации, в которой работает аналитик по разметке.

В состав отчета рекомендуется включить следующие разделы:

- Общая статистика – количество рассмотренных строк кода, выявленное количество детекторов и уязвимостей;

- Подробное описание результатов по каждому детектору: название-описание детектора, маркеры (срабатывания), какие являются истинными, а какие ложноположительными, основания для выработки заключения по маркеру, рекомендации по исправлению для истинных срабатываний.
- Общая статистика и таблица найденных детекторов согласно методике. Возможно использование для этих целей отчетов, выдаваемых Svacer. Рекомендуется передать в составе отчета таблицы, вырабатываемые Svacer до и после разметки.

4. Примеры описания правил анализа детекторов для использования аналитиками по разметке

Методика разметки, включенная в состав системы [14], содержит подробное описание всех типов детекторов, которые представлены в Таблице 1 и обнаруживаются анализатором Svace. По каждому детектору дается описание того, в чем суть выявленной угрозы, поясняет возможные способы исправления уязвимости, но оставляет право принятия решения об исправлении за разработчиком. При этом аналитик по разметке выдает квалифицированные рекомендации по исправлению кода. Наличие такого описания позволяет снизить начальные требования к уровню знания языка программирования аналитиком по разметке. Соответственно после нескольких разметок такой аналитик начинает лучше понимать срабатывания и выявленные дефекты и глубже погружаться в свойства языка, повышать скорость и эффективность достижения высокого качества разметки. Последнее может быть полезно для работников локальных служб информационной безопасности.

На рисунках 2-5 приведены примеры детекторов для программ на языке Java (рис. 2 и 3) и Python (рис. 4 и 5).

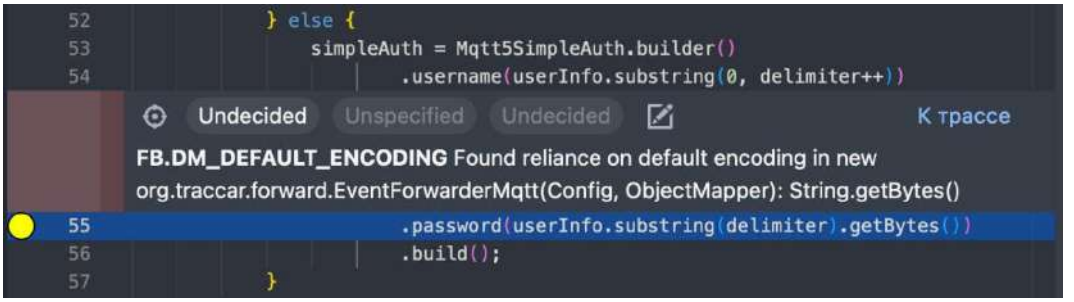


Рис. 2. Детектор FB.DM_DEFAULT_ENCODING в интерфейсе Svacer.
Fig. 2. FB.DM_DEFAULT_ENCODING detector in the Svacer interface.

Детектор "FB.DM_DEFAULT_ENCODING" определяет ошибку – вызов метода, который выполняет преобразование байта в строку (или строку в байт) и предполагает, что кодировка платформы по умолчанию подходит. Это приведет к тому, что поведение приложения будет различаться на разных платформах. Используйте альтернативный API (application programming interface — программный интерфейс приложения) и явно укажите имя набора символов или объект Charset.

Рис. 3. Детектор FB.WA_NOT_IN_LOOP в интерфейсе Svacer.
Fig. 3. FB.WA_NOT_IN_LOOP detector in the Svacer interface.

Метод детектора "FB.WA_NOT_IN_LOOP" содержит вызов процедуры java.lang.Object.wait(), который не находится в цикле. Если монитор используется для нескольких условий, то ожидаемое событие может быть заменено иным, включенным в монитор. При необходимости отработки ожидаемого условия рекомендуется включить java.lang.Object.wait() в цикл для гарантированной отработки требуемой конструкции.

Рис. 4. Детектор CATCH.NO_BODY в интерфейсе Svacer.
Fig. 4. CATCH.NO_BODY detector in the Svacer interface.

Детектор "CATCH.NO_BODY" возникает, когда в блоке try-except исключение перехватывается, но сам блок except остается пустым или не содержит кода для обработки исключения. Может возникнуть ситуация, когда исключение будет перехвачено, но не будет обработано.

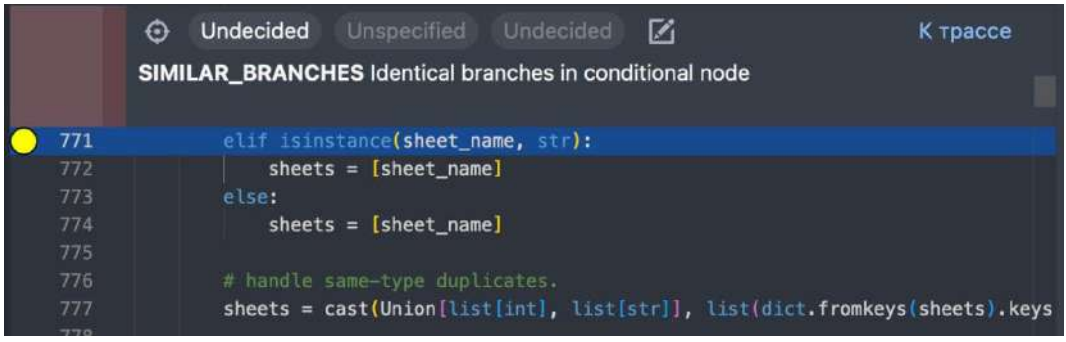


Рис. 5. Детектор SIMILAR_BRANCHES в интерфейсе Svacer.
Fig. 5. SIMILAR_BRANCHES detector in the Svacer interface.

Детектор "SIMILAR_BRANCHES" возникает, когда в условном операторе (if-else) обнаруживаются две идентичные ветви кода, то есть в обеих ветвях выполняется одно и то же действие. Это может указывать на ненужное или избыточное повторение кода, и такая конструкция может быть улучшена или упрощена.

5. Связь с CWE

Разработанная методика имеет потенциал к развитию за счет систематизации детекторов как уязвимостей. На данный момент в методике (и в Svace) поддерживается классификация отечественного регулятора ФСТЭК. С позиций развития методики основной идеей является конвертируемость результатов в различные классификации уязвимостей. Одной из наиболее распространённых и общепринятых в мире является классификация CWE (Common Weakness Enumeration [15] – классификация недостатков безопасности в программном и аппаратном обеспечении. Она представляет собой иерархический словарь, предназначенный для разработчиков и специалистов по обеспечению безопасности ПО [8]. Основная цель CWE – предотвращать возникновение уязвимостей за счёт обучения специалистов наиболее распространённым типам уязвимостей и за счет этого – предотвращение их возникновения в программах.

Совместное использование классификаций - ФСТЭК и CWE, (а впоследствии и CVE) позволит повысить процент выявляемых типов уязвимостей, которые сегодня не поддерживаются при использовании классификации регулятора, но могут быть выявлены при совместном использовании двух и более международных классификаций. Цель использования международных классификаций - конвертируемость результатов в различные метрики оценки качества поиска уязвимостей. В настоящее время ведутся работы по сопоставлению классификаций ФСТЭК и CWE, что позволит улучшить понимание выявляемых уязвимостей, расширить состав выявляемых уязвимостей и упростит работу по их исправлению.

6. Заключение

По итогам проведенных работ получены следующие результаты:

- Разработан процесс выявления уязвимостей аналитиком по разметке с использованием инструментальных средств Svace/Svacer.
- Разработана методика разметки программ с применением документа «Методика статического анализа для программ на языке» – то есть методика поиска уязвимостей и недекларированных возможностей в программах на пяти языках программирования (C/C++, Java, Python, Go). Она ориентирована на помощь экспертам в выявлении неочевидных моментов и в проверке подлинности найденных угроз, при использовании в качестве статического анализатора Svace и снижает квалификационные требования к аналитикам по разметке, одновременно являясь учебником по проведению разметки. Данная методика включена в состав Автоматизированной Системы Центр кибербезопасности.
- Ключевым направлением развития этих работ является создание плагина для продукта Svacer. Плагин позволит для каждого рассматриваемого детектора по запросу аналитика получить:
  - описание сути найденного детектора, предполагаемой уязвимости или НДВ (если она обнаружена),
  - причины, по которой эта конструкция в данном языке и в текущих условиях применения конструкции скорее всего является уязвимостью,
  - в случае возможных нескольких вариантов уязвимости данной конструкции – получить примеры правильных конструкций.

При реализации плагина будет реализована возможность совместного использования классификаций уязвимостей ФСТЭК и CWE.

Список литературы / References

[1]. "Белеванцев А. (ИСП РАН) для форума "Russia DevOps Report - 2023". [Электронный ресурс]. – 2023 – URL: https://russiadevopsreport.ru/. – (Дата обращения: 13.10.2024)

[2]. ГОСТ Р 71207-2024. Защита информации. Разработка безопасного программного обеспечения. Статический анализ программного обеспечения. Общие требования.

[3]. PVS-Studio: Ложноположительные срабатывания статического анализатора кода. [Электронный ресурс]. – 2021 – URL: https://pvs-studio.ru/ru/blog/terms/6461/. – (Дата обращения: 03.11.2024)

[4]. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН. 2014;26(1): с.231-250. / Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. Svace static analyzer for searching for defects in program source code. Proceedings of the Institute of System Programming of the Russian Academy of Sciences. 2014;26(1): p.231-250. DOI: https://doi.org/10.15514/ISPRAS-2014-26(1)-7.

[5]. Белеванцев А., Аветисян А. Многоуровневый статический анализ для поиска закономерностей ошибок и дефектов в исходном коде. В: Петренко А., Воронков А. (ред.) Перспективы системной информатики. PSI 2017. Конспекты лекций по информатике, том 10742, стр. 28–42. /Belevantsev, A., Avetisyan, A. Multi-level Static Analysis for Finding Error Patterns and Defects in Source Code. В: Petrenko, A., Voronkov, A. (ред.) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science, vol 10742, p. 28–42. Springer, Cham, 2018, DOI:10.1007/978-3-319-74313-4_3.

[6]. Positive Technologies: Positive Technologies: раскрытие уязвимостей и опыт взаимодействия исследователей и вендоров в 2022–2023 годах. [Электронный ресурс]. – 2024 – URL: https://www.ptsecurity.com/ru-ru/research/analytics/vulnerability-disclosure-and-researcher-vendor-interaction-experience-in-2022-2023/. – (Дата обращения: 04.11.2024)

[7]. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.

[8]. ГОСТ Р 56939–202X. Защита информации. Разработка безопасного программного обеспечения. Общие требования.

[9]. ГОСТ Р ИСО/МЭК 27034. Информационная технология. Методы и средства обеспечения безопасности. Безопасность приложений. Часть 1. Обзор и общие понятия.

[10]. ГОСТ Р 50922-2006. Защита информации. Основные термины и определения.

[11]. ГОСТ Р 58412-2019. Защита информации. Разработка безопасного программного обеспечения. Угрозы безопасности информации при разработке программного обеспечения.

[12]. КонсультантПлюс: (ФСТЭК) Общая характеристика уязвимостей прикладного программного обеспечения. [Электронный ресурс]. – 2008 – URL: https://www.consultant.ru/document/cons_doc_LAW_99662/2da3bb1b6c61f5acbbefcb29ae7dc99615ce0d05/.– (Дата обращения: 20.10.2024)

[13]. PVS-Studio: Сортировка предупреждений статических анализаторов по приоритету при поиске и исправлении программных ошибок. [Электронный ресурс]. – 2016 – URL: https://habr.com/ru/companies/pvs-studio/articles/305532/. – (Дата обращения: 03.11.2024)

[14]. Б. Позин. Автоматизация и экономика для обеспечения жизненного цикла безопасного ПО., Информационная безопасность, №4, 2024, с.60/ B.Pozin. Automation and Economics for Secure Software Life Cycle Assurance., Information Security, No.4, 2024, p.60.

[15]. Common Weakness Enumeration [Электронный ресурс]. – 2023 – URL: https://cwe.mitre.org/about/new_to_cwe.html. – (Дата обращения: 02.11.2024)

Информация об авторах / Information about authors

Борис Аронович ПОЗИН – доктор технических наук, профессор, главный научный сотрудник Института системного программирования им. В.П. Иванникова РАН, профессор базовой кафедры ЗАО ЕС-лизинг в МИЭМ НИУ ВШЭ, технический директор ЗАО ЕС-лизинг. Сфера научных интересов: программная инженерия, системы обеспечения жизненного цикла доверенного программного обеспечения, автоматизированное тестирование программ.

Boris Aronovich POZIN – Dr. Sci. (Tech.), Professor, Chief Researcher at the Ivannikov Institute for System Programming of the Russian Academy of Sciences, Professor of the Basic Department of CJSC EC-Leasing at the Higher School of Economics, Technical Director of CJSC EC-Leasing. Research interests: software engineering, life cycle ensuring systems for trusted software, automated software testing.

Полина Андреевна БОРОДУШКИНА – студентка третьего курса НИУ ВШЭ Московского институт электроники и математики им. А.Н. Тихонова департамента прикладной математики. Сфера научных интересов: проектирования безопасного ПО с применением SAST, анализ существующих угроз и их классификации на языках программирования C/C++.

Polina Andreevna BORODUSHKINA – third-year student of the National Research University Higher School of Economics, A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Applied Mathematics. A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Applied Mathematics. Her research interests are study of secure software design using SAST, analysis of existing threats and their classification in C/C++ programming languages.

Дмитрий Антонович КОРОТКОВ – студент четвертого курса НИУ ВШЭ Московского институт электроники и математики им. А.Н. Тихонова департамента электронной инженерии. Сфера научных интересов: проектирование безопасного ПО с применением Статического анализа (SAST), верификация результатов статического анализа, анализ существующих угроз и их классификации на языке программирования Java.

Dmitry Antonovich KOROTKOV – fourth-year student of the National Research University Higher School of Economics, A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Electronic Engineering. A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Electronic Engineering. Research interests: design of secure software using Static Analysis (SAST), verification of static analysis results, analysis of existing threats and their classification in Java programming language.

Михаил Александрович ФЕДОРОВ – студент четвертого курса НИУ ВШЭ Московского институт электроники и математики им. А.Н. Тихонова департамента электронной инженерии. Сфера научных интересов: проектирование безопасного ПО с применением Статического (SAST) и Композиционного (SCA) анализов, разработка ПО в сфере фронт-энд, проектирование безопасной архитектуры Операционных систем.

Michael Alexandrovich FEDOROV – fourth-year student of the National Research University Higher School of Economics, A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Electronic Engineering. A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Electronic Engineering. Research interests: design of secure software using Static (SAST) and Compositional (SCA) analyses, front-end software development, design of secure architecture of Operating Systems.

Айнур Фуатович МУРАТОВ – студент третьего курса НИУ ВШЭ Московского институт электроники и математики им. А.Н. Тихонова департамента компьютерной инженерии. Сфера научных интересов: изучение проектирования безопасного ПО с применением SAST, анализ существующих угроз и их классификации на языке программирования Go.

Aynur Fuatovich MURATOV – third year student of the National Research University Higher School of Economics, A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Computer Engineering. A.N. Tikhonov Moscow Institute of Electronics and Mathematics, Department of Computer Engineering. Research interests: studying secure software design using SAST, analyzing existing threats and their classification in Go programming language.

DOI: 10.15514/ISPRAS-2025-37(1)-8

CC BY

САПР для удаленного высокоуровневого моделирования СтнК

А.А. Американов, ORCID: 0000-0002-5970-2125 <aamerikanov@hse.ru>
Л.Г. Евтушенко, ORCID: 0000-0003-1261-9735 <levtushenko@hse.ru>
В.В. Зунин, ORCID: 0000-0002-9117-4879 <vzunin@hse.ru>
В.М. Винарский, ORCID: 0009-0005-5319-6304 <vmvinarskiy@edu.hse.ru>

Национальный исследовательский университет «Высшая школа экономики», Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Статья посвящена описанию процесса создания новой архитектуры САПР для высокоуровневого моделирования СтнК, а также удаленному маршруту проектирования СтнК. В работе проведен анализ основных этапов проектирования СтнК, в результате чего продемонстрирована высокая значимость высокоуровневого моделирования и его влияние на весь процесс проектирования. Также рассмотрена возможность проведения высокоуровневого моделирования в удаленном формате с использованием клиент-серверной архитектуры, предложенной САПР. Рассмотрен процесс удаленного проектирования СтнК с использованием предложенного САПР и удаленных стендов с отладочными платами ПЛИС.

Ключевые слова: система автоматизированного проектирования (САПР); высокоуровневое имитационное моделирование; сеть на кристалле (СтнК); проектирование СтнК; удаленные стенды.

Для цитирования: Американов А.А., Евтушенко Л.Г., Зунин В.В., Винарский В.М. САПР для удаленного высокоуровневого моделирования СтнК. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 133–144. DOI: 10.15514/ISPRAS–2025–37(1)–8.

Благодарности: Публикация подготовлена в ходе проведения исследования (№ 24-00-012 «Удаленная лаборатория для работы со встраиваемыми системами») в рамках Программы «Научный фонд Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ)».

CAD for Remote High-Level Modeling of NoC

A.A. Amerikanov, ORCID: 0000-0002-5970-2125 <aamerikanov@hse.ru>
L.G. Evtushenko, ORCID: 0000-0003-1261-9735 <levtushenko@hse.ru>
V.V. Zunin, ORCID: 0000-0002-9117-4879 <vzunin@hse.ru>
V.M. Vinarskii, ORCID: 0009-0005-5319-6304 <vmvinarskiy@edu.hse.ru>
HSE University,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. The paper is devoted to the description of the process of developing a new CAD architecture for high-level modeling of NoC, as well as the remote flow of NoC design. The paper analyzes the main stages of NoC design and demonstrates the high importance of high-level modeling and its impact on the entire design process. Also, the possibility of conducting high-level modeling in a remote format using the client server architecture of the CAD is considered. The process of remote design of NoC using the proposed CAD system and remote testbed with FPGA debug boards is demonstrated.

Keywords: computer-aided design (CAD); high-level modeling; network-on-chip (NoC); NoC design; remote testbed.

For citation: Amerikanov A.A., Evtushenko L.G., Zunin V.V., Vinarskii V.M. CAD for remote high-level modeling of NoC. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 1, 2025, pp. 133-144. DOI: 10.15514/ISPRAS-2025-37(1)-8.

Acknowledgements. The publication was prepared within the framework of the Academic Fund Program at HSE University (grant № 24-00-012 Remote laboratory for working with embedded systems).

1. Введение

С каждым годом растет сложность решаемых задач и объем информации, которую требуется обрабатывать, а значит растут и требования, предъявляемые к производительности систем на кристалле [1]. Задачи, требующие высокой вычислительной производительности, и задачи по работе с большим потоком данных не всегда могут быть эффективно решены с помощью однопроцессорных систем [2]. Увеличение плотности размещения транзисторов на чипе становится все более сложным, в связи с чем происходит эволюция вычислительных систем в сторону многоядерности, многопоточности и использования специализированных вычислительных ядер и ускорителей [3, 4]. Чип WSE2 от компании Cerebras [5] является ярким примером экстенсивного увеличения количества ядер на одном чипе; он выполнен по 7 нанометровому техпроцессу и состоит из 850000 вычислительных узлов. Также существует тенденция к замене громоздкой CISC архитектуры [6] на сети из процессорных ядер, разработанных на основе RISC архитектуры [7]. Из этого следует повышение требований к подсистеме связи для объединения множества гетерогенных ядер в одну систему – сеть на кристалле (СтнК) [8].

Процесс проектирования СтнК можно разделить на несколько основных последовательных этапов [9]: подготовка технического задания, предварительное проектирование, высокоуровневое и низкоуровневое моделирование, прототипирование или косимуляция, производство. Для каждого этапа обычно используются различные специализированные системы автоматизации проектирования (САПР), но другим возможным решением является применение единого САПР для всего процесса в целом.

Этап высокоуровневого моделирования требуется, чтобы выделить ограниченное количество подходящих для дальнейшего проектирования наборов параметров и характеристик сети, заданных на стадиях составления технического задания и проектирования СтнК. Цена ошибки на этапе высокоуровневого моделирования очень высока, так как низкоуровневое моделирование представляет собой гораздо более длительный и трудоемкий процесс. Например, высокоуровневое моделирование сети на 100 узлов с использованием

высокоуровневой модели OCNS (On-Chip Network Simulator) [10] может занимать несколько минут, в то время как низкоуровневое моделирование этой сети в программе моделирования, например ModelSim [11], может занять несколько дней.

Обычно на этапе высокоуровневого моделирования применяются имитационные модели [12]. Имитационная модель СтнК – это модель, в которой описаны основные моделируемые элементы сети и заданы правила взаимодействия между ними. Как правило, высокоуровневые модели используются для описания процесса передачи данных в СтнК и получения предварительных оценок ее характеристик при заданных параметрах. Процесс передачи данных является критически важным элементом функционирования сети. На этапе высокоуровневого моделирования точностью представления некоторых аспектов функционирования сети можно пренебречь, увеличив скорость расчета характеристик. В низкоуровневом моделировании напротив, учитывается как можно больше аспектов функционирования сети, например, учитывается реализация сложно-функциональных блоков, чтобы получить более точный результат моделирования.

Следует также отметить, что различные высокоуровневые модели решают разные, зачастую узкоспециализированные, задачи. Существует множество различных средств автоматизации моделирования СтнК и других вспомогательных средств, созданных разными разработчиками.

В проектировании нет единого подхода для объединения различных методов автоматизации проектирования СтнК на уровне архитектуры. Это позволило бы реализовать сквозное проектирование СтнК, используя разные модели и средства на разных этапах разработки, обеспечивая их совместимость через универсальные интерфейсы и форматы данных. Также обычно передача данных между этапами высокоуровневого и низкоуровневого моделирования в основном выполняется вручную, что приводит к увеличению трудозатрат и ошибок [13]. Кроме того необходимо, чтобы была возможность проведения удаленного запуска сценариев моделирования на мощной вычислительной машине, чтобы не разворачивать средства моделирования на разных машинах. Таким образом, разработка единых средств анализа результатов высокоуровневого моделирования и автоматизации процесса моделирования являются важной и актуальной научно-практической задачей, которой посвящена данная статья.

Далее в разделе 2 сформулирована проблема, решаемая в данном исследовании. В разделе 3 проведен анализ цикла проектирования СтнК, приведена типовая схема проектирования СтнК, обоснована важность этапа высокоуровневого моделирования. В разделе 4 проведен краткий анализ высокоуровневых моделей СтнК. В разделе 5 описан процесс создания САПР для высокоуровневого моделирования СтнК, приведены схемы структуры и работы САПР и рассмотрен перенос предложенного САПР высокоуровневого моделирования СтнК на клиент-серверную архитектуру. Такой подход позволит выполнять этап высокоуровневого моделирования с использованием удаленных стендов, с возможностью последующего прототипирования и косимуляции.

2. Высокоуровневое моделирование СтнК

В настоящее время существует множество высокоуровневых моделей, предлагаемых к практическому использованию для моделирования СтнК. Авторами работы исследованы различные реализации высокоуровневых моделей, которые могут применяться для СтнК [14–17]. Они отличаются по множеству различных параметров, например, по функциональному назначению, типам трафика, областям применения. В настоящее время кажущееся многообразие не привело к появлению какой-либо универсальной модели, которая могла бы учитывать все возможные варианты конфигураций СтнК, и выполняла бы расчет всех возможных параметров сети. Обычно модель подбирается в зависимости от технических требований и решений, принятых на этапе проектирования. При этом большинство средств

моделирования СтнК не имеют каких-либо встроенных инструментов автоматизации расчетов (например, возможности запуска нескольких моделей одновременно или автоматического подбора параметров в зависимости от целей моделирования).

Из сказанного следует необходимость создания новых инструментов и методов автоматизации высокоуровневого моделирования СтнК, а также разработки единых средств автоматизации проектирования для проведения исследования характеристик СтнК, то есть существует потребность в создании САПР, которая имела бы возможность объединять в себе и сравнивать несколько высокоуровневых моделей и результаты их работы, тем самым увеличивая точность моделирования. Также объединение множества высокоуровневых моделей в единой САПР позволит сократить время моделирования путем применения методов оптимизации при поиске различных параметров СтнК.

Таким образом, проблема, решаемая в данном исследовании, заключается в существовании множества разнородных высокоуровневых моделей СтнК и отсутствии САПР, которая позволила бы объединить их в рамках единой среды проектирования, провести оценку достоверности результатов, а также сохранить результаты предыдущих циклов моделирования.

3. Анализ классического цикла проектирования сетей на кристалле

Для проектирования сложных многоядерных вычислительных систем все чаще используют архитектурные решения из области СтнК. СтнК – многопроцессорные системы на кристалле, где роль вычислительных узлов выполняют сложно функциональные блоки (СФ-блоки), которые связаны короткими соединениями на чипе для обмена информацией. На рис. 1 изображена структура СтнК.

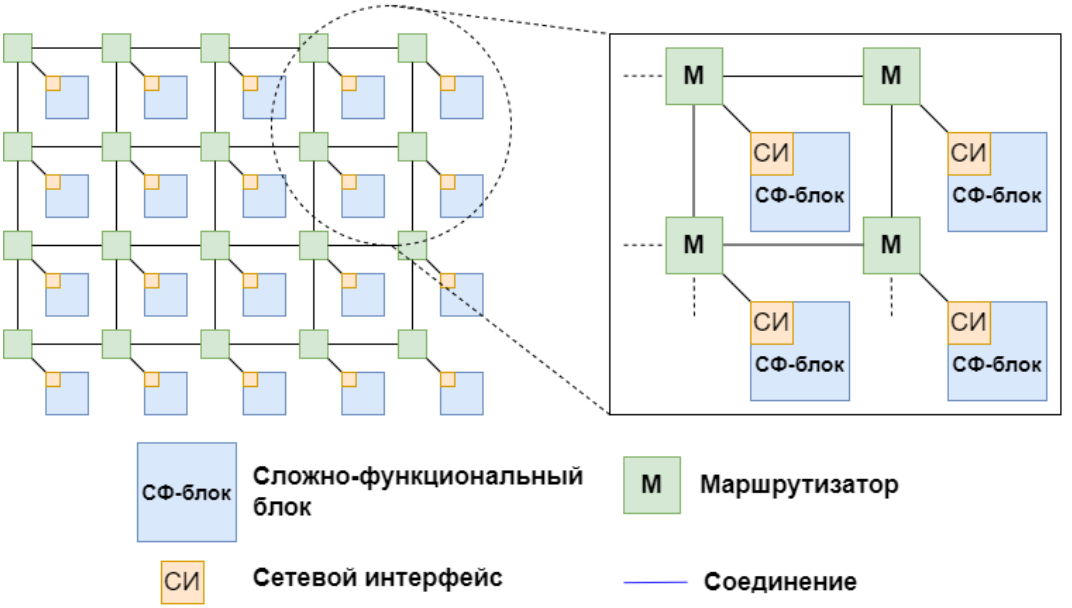


Рис. 1. Структура СтнК.
Fig. 1. NoC structure.

Особенностями СтнК являются ограниченность набора средств автоматизации основных этапов проектирования, из-за малого количества специализированных САПР для проектирования СтнК, и невозможность полного отображения алгоритмов, методов и протоколов маршрутизации, применяемых в классических сетях из-за ограниченных ресурсов.

3.1 Типовая схема проектирования сетей на кристалле

На рис. 2 приведена типовая схема проектирования СтнК. Сплошными стрелками на схеме показаны переходы к следующим этапам проектирования. Ошибки на разных этапах проектирования вынуждают разработчика возвращаться на предыдущий этап, что показано пунктирными стрелками на схеме.

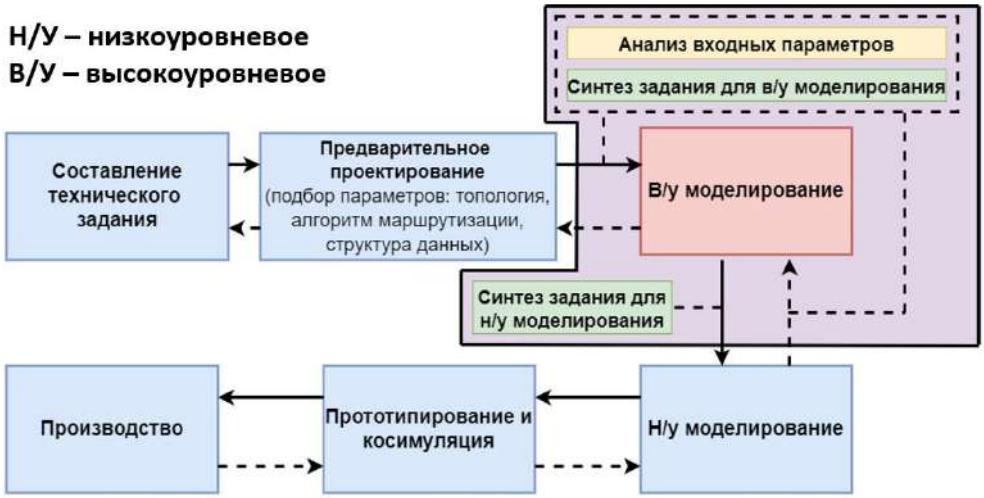


Рис. 2. Типовая схема проектирования СтнК.
Fig. 2. Typical scheme for NoC design.

На этапе подготовки технического задания, состоящего из определения желаемых характеристик СтнК, необходимо учитывать следующие параметры:

1) ограничения аппаратных ресурсов;
2) формат данных, передаваемых в СтнК между вычислительными ядрами;
3) область применения СтнК.

На этапе предварительного проектирования происходит:

1) выбор топологии;
2) выбор типа маршрутизации (детерминированная/адаптивная);
3) выбор типа данных, передаваемых между узлами;
4) выбор типа маршрутизаторов и вычислительных ядер.

Важным этапом проектирования является высокоуровневое моделирование. Данный этап позволяет отобрать ограниченное количество подходящих для дальнейшего проектирования наборов параметров и характеристик сети, выбранных на стадии проектирования СтнК.

Основными оцениваемыми параметрами являются:

1) максимальный/минимальный путь пакета;
2) отказоустойчивость (для адаптивных алгоритмов);
3) гарантированное время доставки пакетов;
4) максимальная пропускная способность сети;
5) устойчивость к блокировкам (дедлокам, лайвлокам);
6) пропускная способность;
7) загруженность буферов;
8) энергопотребление.

Ввиду большого разнообразия входных и выходных характеристик, определенная высокоуровневая модель может не иметь возможности расчета некоторых параметров. Тогда разработчик вынужден использовать для высокоуровневого моделирования несколько моделей, что увеличивает время моделирования, т.к. каких-либо инструментов позволяющих провести совместное моделирование СтнК несколькими моделями в рамках единой САПР не существует.

Низкоуровневое моделирование является более тщательной проверкой выходных характеристик СтнК, чем высокоуровневое моделирование и позволяет оценить:

1) затраты аппаратных ресурсов;
2) более точно, чем при высокоуровневом моделировании, пропускную способность (и другие высокоуровневые показатели);
3) наличие ошибок при маршрутизации.

Несмотря на то, что входные параметры СтнК определяются на этапе проектирования и являются одинаковыми для низкоуровневого и высокоуровневого моделирования, при исследовании не было найдено САПР, которая способна формировать задания для низкоуровневого этапа моделирования, основываясь на данных, введенных разработчиком для высокоуровневого этапа моделирования.

Этап прототипирования на программируемых логических интегральных схемах (ПЛИС) либо косимуляция помогает:

1) оценить аппаратные ресурсы сети;
2) оценить частоту работы сети;
3) определить работоспособность СтнК (нагрузочное тестирование).

Этап производства является финальным этапом, после которого СтнК реализуется на ПЛИС или производится в виде заказной микросхемы.

Данная работа относится к этапу высокоуровневого моделирования. На этом этапе с помощью специальных высокоуровневых моделей, по выбранным на этапе проектирования характеристикам и параметрам, производится подсчет выходных характеристик той или иной сети.

Высокоуровневое моделирование является важным этапом проектирования СтнК, а цена ошибки на этом этапе очень высока, поскольку приводит к избыточному проведению низкоуровневого моделирования, которое на несколько порядков более затратно по времени, чем высокоуровневое моделирование. Например, из работы [10] следует, что высокоуровневое моделирование сети на 100 узлов с помощью модели OCNS (On-Chip Network Simulator) [18] занимает несколько минут, в то время как низкоуровневое моделирование той же сети с использованием Netmaker [19] – несколько дней.

4. Анализ разнообразия высокоуровневых моделей сетей на кристалле

Теоретические основы СтнК за последние десятилетия развивается очень интенсивно. Если ранее ключевыми были книги Axel Jantsch [20], Luca Benini [21], William James Dally [22], а моделей СтнК в открытом доступе были единицы [19], то сейчас количество публикаций и открытых разработок по тематике проектирования СтнК стало больше. В результате современному исследователю становится достаточно проблематично ориентироваться в различных идеях и подходах, предлагаемых другими авторами. При этом обзорных работ, где были бы собраны в сжатом и структурированном виде современные достижения в СтнК довольно мало. Есть ряд университетских курсов [23] и статей [24,25], дающих общий обзор тематики СтнК. Подробных обзоров из каких-то отдельных областей проектирования СтнК, как, например, это сделано в статье [26] в виде классификации алгоритмов маршрутизации,

также мало. Это касается и попыток классификации высокоуровневых моделей СтнК, которая представлена в работе [16]. Поэтому требуются проведение обзора и классификации подходов, используемых на этапах проектирования СтнК.

Для того, чтобы подготовить обзор в области высокоуровневого моделирования СтнК, было исследовано более 100 моделей. В основном были рассмотрены модели собранные в источнике [17]. В нем авторы представили результаты исследования в виде таблицы, что позволило в одном месте в компактном и структурированном виде собрать информацию по предметной области высокоуровневых моделей СтнК.

Проведенный обзор наиболее известных высокоуровневых моделей СтнК показал, что существуют различные модели, которые в основном решают частные задачи, нет универсальных моделей, все они не стандартизированы и зачастую не совместимы друг с другом. Для того, чтобы провести моделирование СтнК с собственной комбинацией параметров топологии, маршрутизации, метода контроля и генерации трафика, арбитража и т.д., необходимо создавать собственную модель или дорабатывать существующую. При этом практически отсутствуют средства автоматизации моделирования, которые бы облегчили обработку результатов моделирования и их верификации, проведение множественного запуска модели с различными параметрами и так далее. Нет средств интеграции моделей между собой. Такое состояние проблемной области определяет необходимость разработки САПР, которая бы позволила решить обозначенные проблемы.

5. Разработка архитектуры САПР для высокоуровневого моделирования СтнК

5.1 Архитектура САПР

Каждая модель СтнК имеет свой набор входных параметров и выходных характеристик. Когда требуется найти множество характеристик, для расчета которых нужно задействовать несколько моделей, разработчику приходится формировать задание на моделирование для каждой модели СтнК по отдельности, а затем собирать результаты моделирования вместе. Это увеличивает временные затраты на высокоуровневое моделирование, а также усложняет анализ полученных результатов и их передачу на следующий этап проектирования СтнК.

В разрабатываемую САПР для моделирования могут быть включены различные модели, благодаря чему, при получении набора параметров САПР может автоматически распределять эти параметры между моделями. Далее рассчитанные характеристики собираются и обрабатываются САПР, и пользователь получает единый подробный отчет. Таким образом можно упростить и автоматизировать процесс исследований характеристик СтнК и расширить список входных параметров и оцениваемых характеристик. Кроме того, можно сократить количество ошибок при оценивании тех характеристик, которые генерируются хотя бы двумя моделями, так как имеется возможность сравнения выходных характеристик обеих моделей.

Для разработки САПР требуется обеспечить:

1) синтез конфигурационных файлов для высокоуровневых моделей;
2) анализ задания пользователя на моделирование и оптимизацию расчетов;
3) синтез отчета с рассчитанными характеристиками;
4) анализ и хранение полученных результатов.

Для реализации САПР предлагается использовать архитектуру, изображенную на рис. 3. Архитектура состоит из ядра САПР, которое выделено сиреневым цветом, GUI для взаимодействия с пользователем, базы данных с результатами моделирования и подключаемых к ядру высокоуровневых имитационных моделей. Данные высокоуровневой модели являются разнородными и набор их неограничен.

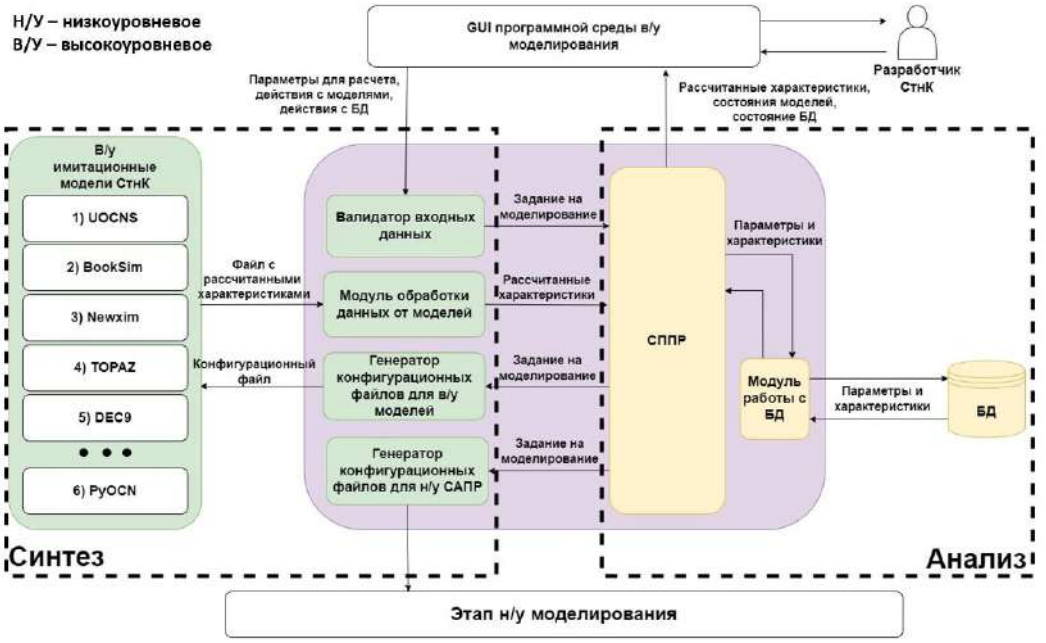


Рис. 3. Архитектура разработанной САПР.
Fig. 3. Architecture of the CAD developed.

Рассмотрим подробнее предлагаемую архитектуру САПР. Архитектура САПР логически разделена на этапы синтеза и анализа. Блок синтеза состоит из валидатора входных данных, генератора конфигурационных файлов для низкоуровневой САПР, модуля обработки получаемых от моделей данных, генератора конфигурационных файлов для высокоуровневых моделей и подключаемых моделей. Блок анализа состоит из системы поддержки принятия решений (СППР) и модуля работы с базой данных (БД).

После ввода пользователем задания, системой производится валидация входных данных и оценка задания СППР. СППР позволяет оценить производилось ли моделирование СтнК данной конфигурации ранее. Если проводилось, то система передает результаты моделирования из базы данных пользователю, если нет, то система помогает пользователю выбрать необходимые методы и высокоуровневые модели для проведения моделирования. Взаимодействие ядра САПР с моделями осуществляется путем передачи конфигурационного файла моделям и файла с рассчитанными характеристиками от модели ядру САПР. За генерацию конфигурационного файла и обработку данных, полученных от модели, отвечают два специализированных блока в ядре САПР.

Также стоит отметить, что, при необходимости, САПР способна сконфигурировать задание на низкоуровневое моделирование для передачи на следующий этап проектирования СтнК.

5.2 Клиент-серверная архитектура САПР

Для повышения эффективности и удобства работы с САПР, целесообразно перевести ее на клиент-серверную архитектуру, которая приведена на рис. 4. В предложенной архитектуре САПР представляет собой распределенную систему с веб-интерфейсом, ядром САПР, базой данных результатов моделирований и моделями СтнК.

Такая архитектура позволяет разработчику СтнК взаимодействовать с системой удаленно через веб-интерфейс. Разработчик вводит необходимые характеристики в веб-интерфейс. Далее, ядро САПР принимает входные характеристики и выполняет ряд задач, включая генерацию конфигурационных файлов для запуска моделей. Высокоуровневые модели могут

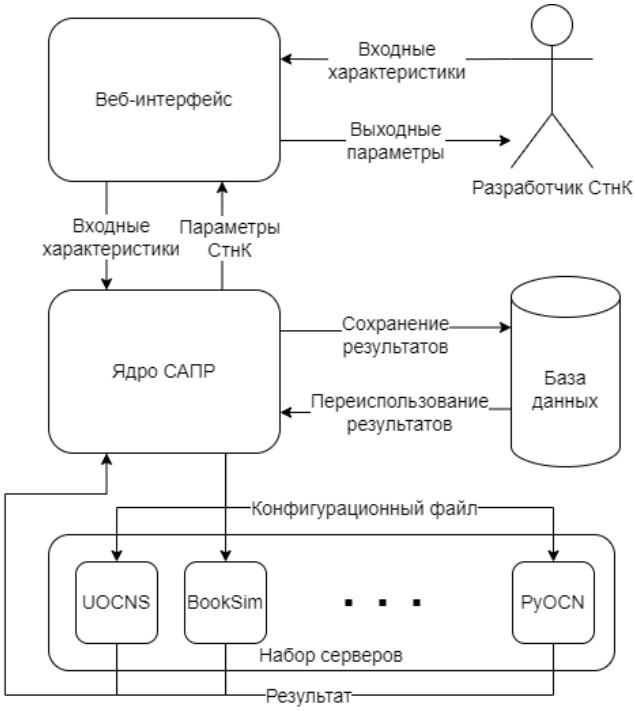


Рис. 4. Клиент-серверная архитектура САПР.
Fig. 4. Client-server CAD architecture.

находиться на отдельных высокопроизводительных серверах. После формирования задания ядро проверяет наличие аналогичных расчетов в базе данных. Если подобное моделирование уже проводилось ранее, ядро САПР запрашивает результаты из базы данных, избегая повторных вычислений. В случае, если заданная конфигурация моделируется впервые, система запускает соответствующую модель на сервере, используя сгенерированный конфигурационный файл. По завершении расчетов результаты работы моделей собираются и агрегируются ядром САПР. Результаты индивидуальных запусков моделей сохраняются в базе данных, и выводятся пользователю через веб-интерфейс. Клиент-серверный подход к организации САПР позволяет реализовать многопользовательскую работу, делая возможным одновременное взаимодействие с САПР нескольких разработчиков через удаленный доступ. Благодаря распределенной архитектуре и возможности применения множества производительных серверов, клиент-серверная архитектура САПР позволит значительно ускорить процесс моделирования СтнК. Также можно применить кеширование результатов работы моделей всех пользователей, которое сократит количество повторных вычислений и тем самым уменьшит время расчета параметров СтнК. Модульная архитектура и централизованное хранение результатов в базе данных позволяют улучшить целостность хранимых данных и обеспечивают возможность масштабирования системы для поддержки новых серверов и моделей по мере необходимости.

5.3 Удаленный маршрут проектирования СтнК

Рассмотрим более подробно типовую схему проектирования СтнК (рис. 2). Все этапы из данной схемы, кроме производства, можно реализовать удаленно, используя специализированные инструменты. Этапы составления технического задания и предварительного проектирования могут быть выполнены на компьютере разработчика.

Этапы высокоуровневого и низкоуровневого моделирования, хотя и могут быть выполнены на компьютере разработчика, требуют больших вычислительных затрат. Предложенная схема удаленного стенда для высокоуровневого моделирования СтнК позволяет решить проблему со слабыми вычислительными мощностями у пользователя. Такая схема также позволяет безопасно сохранять большое количество результатов проведенных пользователями моделирований (например, с помощью системы облачного хранения Synology Drive). Данную базу можно использовать для прогнозирования результатов расчетов или выводить пользователю результаты без необходимости запуска симулятора (если они уже есть в базе). Подобную архитектуру можно применить и для этапа низкоуровневого моделирования.

Этап прототипирования и симуляции можно проводить на стендах для удаленной работы со специализированным оборудованием. Например, осуществлять прототипирование разработанной СтнК на удаленных стендах для работы с платами ПЛИС [27,28]. Также существует подход, при котором можно разделить реализацию СтнК на несколько ПЛИС [29,30], что позволит выполнять прототипирование СтнК с большим количеством ядер.

Таким образом маршрут проектирования СтнК может быть полностью удаленным, включая моделирование СтнК на мощных вычислительных ресурсах с их дальнейший косимуляцией и прототипированием на удаленных стендах.

6 Заключение

В статье проведен анализ классического цикла проектирования СтнК, выделены его основные этапы, представлена схема с этапами проектирования. Предложена архитектура САПР для моделирования СтнК, а также описана архитектура клиент-серверной реализации САПР, которая позволит выполнять моделирование СтнК удаленно. Предложен маршрут удаленного проектирования СтнК с возможностью высокоуровневого моделирования, косимуляции и прототипирования.

Список литературы / References

[1]. Shalf J. The future of computing beyond Moore’s Law // Philos. Trans. R. Soc. A. The Royal Society Publishing, 2020, vol. 378, No 2166, p. 20190061.

[2]. Shan Z. et al. Object Detection Based On Multi-Process // 2022 IEEE International Conference on Networking, Sensing and Control (ICNSC). IEEE, 2022, pp. 1–5.

[3]. Paul K., Balakrishnan M., Lavenier D. Hardware acceleration of de Novo genome assembly // Int. J. Embed. Syst. Inderscience Publishers, 2017, vol. 9, No 1.

[4]. Jia Z. et al. Dissecting the graphcore IPU architecture via microbenchmarking // arXiv Prepr. arXiv1912.03413. 2019.

[5]. Moore S.K. Cerebras’ New Monster AI Chip Adds 1.4 Trillion Transistors – IEEE Spectrum // Cerebras’ New Monster AI Chip Adds 1.4 Trillion Transistors. 2021. (online). https://spectrum.ieee.org/tech-talk/semiconductors/processors/cerebras-giant-ai-chip-now-has-a-trillions-more-transistors?utm_source=techalert&utm_medium=email&utm_campaign=techalert-04-22-21&utm_content=httpsspectrumieeeorgtechtalksemiconductorsprocessor, accessed 05.11.2024.

[6]. He Y., Chen X. Survey and Comparison of Pipeline of Some RISC and CISC System Architectures // 2023 8th International Conference on Computer and Communication Systems (ICCCS). IEEE, 2023, pp. 785–790.

[7]. Waterman A. et al. The RISC-V instruction set manual, volume I: User-level ISA, version 2.0 // EECS Dep. Univ. California, Berkeley, Tech. Rep. UCB/EECS-2014-54, 2014.

[8]. Bjerregaard T., Mahadevan S. A survey of research and practices of Network-on-chip // ACM Comput. Surv. 2006, vol. 38, No 1, pp. 1–51. DOI: 10.1145/1132952.1132953.

[9]. Sherwani N.A. Algorithms for VLSI physical design automation. Springer Science & Business Media, 2012.

[10]. Romanov A., Ivannikov A. SystemC Language Usage as the Alternative to the HDL and High-level Modeling for NoC Simulation // Int. J. Embed. Real-Time Commun. Syst. 2018. DOI: 10.4018/IJERTCS.2018070102.
[11]. Jain A. et al. Scalable design and synthesis of 3D mesh network on chip // Proceeding of International Conference on Intelligent Communication, Control and Devices: ICICCD 2016. Springer, 2017, pp. 661 – 666.
[12]. Koohi S. et al. High-level modeling approach for analyzing the effects of traffic models on power and throughput in mesh-based NoCs // 21st international conference on VLSI design (VLSID 2008). IEEE, 2008, pp. 415–420.
[13]. Czaja S.J., Nair S.N. Human factors engineering and systems design // Handb. Hum. factors Ergon. Wiley Online Library, 2012, pp. 38–56.
[14]. Romanov A.Y. et al. Development of routing algorithms in networks-on-chip based on two-dimensional optimal circulant topologies // Heliyon. Elsevier Ltd, 2020, vol. 6, No 1, p. e03183. DOI: 10.1016/j.heliyon.2020.e03183.
[15]. Romanov A.Y., Stepanov M.A. UOCNS-SE: Universal On-Chip Network Simulator Server (online). https://github.com/RomeoMe5/UOCNS-SE.
[16]. Prilepko P.M., Romanov A., Lezhnev E.V. Modification of a High-Level NoCModel 2.0 for Modeling Networks-on-Chip with Circulant Topologies // Probl. Adv. micro- Nanoelectron. Syst. Dev. 2020. P. 23–30. DOI: 10.31114/2078-7707-2020-4-23-30.
[17]. Romanov A.Y., Opekunova A.A. NoC simulators comparison. 2020. https://github.com/RomeoMe5/NoC_simulators_comparison, accessed 05.11.2024.
[18]. Romanov A.Y., Romanova I.I. Use of irregular topologies for the synthesis of networks-on-chip // 2015 IEEE 35th International Conference on Electronics and Nanotechnology (ELNANO). IEEE, 2015, pp. 445–449. DOI: 10.1109/ELNANO.2015.7146927.
[19]. Romanov O., Lysenko O. The comparative analysis of the efficiency of regular and pseudo-optimal topologies of networks-on-chip based on Netmaker // 2012 Mediterranean Conference on Embedded Computing (MECO). IEEE, 2012, pp. 13–16.
[20]. Jantsch A., Tenhunen H. Networks on Chip // Networks on Chip / ed. Jantsch A., Tenhunen H. Boston, MA: Springer US, 2003. 312 p. DOI: 10.1007/b105353.
[21]. De Micheli G., Benini L. Networks on Chips. First Edition / ed. Benini L. Morgan Kaufmann, 2006. 408 p.
[22]. Dally W.J., Towles B.P. Principles and Practices of Interconnection Networks. Elsevier, 2003. 581 p.
[23]. Balasubramonian R. CS6810 Lectures (online). Доступно по ссылке: http: https://www.youtube.com/playlist?list=PL8EC1756A7B1764F6, accessed 05.11.2024.
[24]. Kakoee M.R., Bertacco V., Benini L. ReliNoC: A reliable network for priority-based on-chip communication // 2011 Design, Automation & Test in Europe. IEEE, 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763112.
[25]. Bertozzi D. et al. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip // IEEE Trans. Parallel Distrib. Syst. 2005, vol. 16, No 2, pp. 113–129. DOI: 10.1109/TPDS.2005.22.
[26]. Gabis A.B., Koudil M. NoC routing protocols – objective-based classification // J. Syst. Archit. Elsevier B.V., 2016, vol. 66–67, pp. 14–32. DOI: 10.1016/j.sysarc.2016.04.011.
[27]. Lorens A., Petukhov G., Romanova I. FPGA-Based Asynchronous Remote Laboratory for Online Learning // 2022 International Russian Automation Conference (RusAutoCon). IEEE, 2022, pp. 623–627. DOI: 10.1109/RusAutoCon54946.2022.9896325.
[28]. Измайлова Л.Г., Белоруков А.М., Романов А.Ю. Дистанционный стенд для синхронной работы с оборудованием на основе ПЛИС // Проблемы разработки перспективных микро- и наноэлектронных систем. 2022, vol. 4, pp. 117–121.
[29]. Romashikhin M.Y., Romanova I.I. Parallel Calculation of π in NoCs Using a Remote Testbed // 2024 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM). IEEE, 2024, pp. 996–1000. DOI: 10.1109/ICIEAM60818.2024.10553762.
[30]. Romanov A., Lerner A., Amerikanov A. Cycle-accurate multi-FPGA platform for accelerated emulation of large on-chip networks // The Journal of Supercomputing. 2024, No 80, pp. 22462–22478. DOI: 10.1007/s11227-024-06306-3.

Информация об авторах / Information about authors

Александр Александрович АМЕРИКАНОВ – кандидат технических наук, доцент национального исследовательского университета «Высшая школа экономики». Сфера

научных интересов: разработка САПР, разработка логических устройств, ПЛИС, сети на кристалле.

Aleksandr Aleksandrovich AMERIKANOV – Cand. Sci (Tech.), Associate Professor at the HSE University. Research interests: CAD development, development of logic devices, FPGA, networks-on-chip.

Лариса Геннадьевна ЕВТУШЕНКО – ассистент национального исследовательского университета «Высшая школа экономики». Сфера научных интересов: разработка логических устройств, ПЛИС.

Larisa Genadevna EVTUSHENKO – Assistant lecturer at the HSE University. Research interests: development of logic devices, FPGA.

Владимир Викторович ЗУНИН – старший преподаватель национального исследовательского университета «Высшая школа экономики». Сфера научных интересов: разработка САПР, разработка логических устройств, ПЛИС.

Vladimir Victorovich ZUNIN – Senior Lecturer at the HSE University. Research interests: CAD development, development of logic devices, FPGA.

Владимир Максимович ВИНАРСКИЙ – студент национального исследовательского университета «Высшая школа экономики». Сфера научных интересов: системы на кристалле, ПЛИС, машинное обучение.

Vladimir Maksimovich VINARSKII – Student at the HSE University. Research interests: SoC, FPGA, machine learning.

DOI: 10.15514/ISPRAS-2025-37(1)-9

CC BY

Глубокое обучение в задаче разработки системы автоматической транскрипции

О.В. Гончарова, ORCID: 0000-0003-1044-6244 <goncharovaov@pgu.ru>

Пятигорский государственный университет,
Россия, 357532, г. Пятигорск, Ставропольский край, пр. Калинина, 9,
Российский университет дружбы народов имени Патриса Лумумбы,
Россия, 117198, г. Москва, ул. Миклухо-Маклая, 6,
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В статье представлена архитектура глубокой нейронной сети для автоматического распознавания фонем в речевом сигнале. Предложенная модель использует комбинацию сверточных и рекуррентных слоев, а также механизм внимания, обогащенный референсными значениями формант гласных фонем. Это позволяет эффективно извлекать локальные и глобальные акустические признаки, необходимые для точного распознавания последовательностей фонем. Особое внимание уделяется проблеме несбалансированности частоты фонем в обучающем наборе данных и способам ее преодоления, таким как аугментация данных и применение взвешенной функции потерь. Представленные результаты демонстрируют работоспособность предложенного подхода, однако указывают на необходимость дальнейшего совершенствования модели для достижения более высоких показателей точности и полноты в задаче распознавания речи.

Ключевые слова: Автоматическое распознавание речи; фонетическая транскрипция; глубокие нейронные сети; форманты.

Для цитирования: Гончарова О.В. Применение глубокого обучения для разработки системы автоматической транскрипции. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 145–158. DOI: 10.15514/ISPRAS–2025–37(1)–9.

Благодарности: Исследование подготовлено в рамках гранта РНФ № 23-28-10124 «Квантитативно-статистическая модель анализа эмоционально-маркированной коммуникации в условиях межэтнических взаимодействий в регионе Кавказские Минеральные Воды».

Deep Learning for an Automatic Transcription System Development

O.V. Goncharova, ORCID: 0000-0003-1044-6244 <goncharovaov@pgu.ru>
Pyatigorsk State University,
9, Kalinin Ave., Pyatigorsk, Stavropol Rg., 357532, Russia,
The Patrice Lumumba Peoples' Friendship University of Russia,
6, Miklukho-Maklaya st., Moscow, 117198, Russia,
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This paper presents a deep neural network architecture for automatic phoneme recognition in speech signals. The proposed model combines convolutional and recurrent layers, as well as an attention mechanism enriched with reference values of vowel formant frequencies. This allows the model to effectively extract local and global acoustic features necessary for accurate phoneme sequence recognition. Particular attention is paid to the problem of imbalanced phoneme frequency in the training dataset and ways to overcome it, such as data augmentation and the use of a weighted loss function. The reported results demonstrate the viability of the proposed approach, but also indicate the need for further model refinement to achieve higher accuracy and recall in the speech recognition task.

Keywords: Automatic speech recognition; phonetic transcription; deep neural networks; formants.

For citation: Goncharova O.V. Deep Learning for the Development of an Automatic Transcription System. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 145-158 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-9.

Acknowledgements. The study was prepared within the framework of the Russian National Science Foundation Grant No. 23-28-10124 "Quantitative statistical model for the analysis of emotionally marked communication in the context of interethnic interactions in the Caucasian Mineral Waters region".

1. Введение

Современные технологии обработки естественного языка (Natural Language Processing, NLP) и распознавания речи являются фундаментальными компонентами в развитии систем взаимодействия человека и машины. С ростом объёмов аудиоинформации и возрастающей потребностью в её автоматической обработке возникает необходимость в высокоточных моделях, способных эффективно распознавать и интерпретировать звуковые данные.

Фонетические транскрипции являются одной из ключевых единиц в области обработки речи и повсеместно используются в лингвистических исследованиях как для научного анализа, так и для проверки гипотез [1-2]. Однако проблемы, связанные с их получением – значительные временные и финансовые затраты, а также ограниченная точность – становятся особенно актуальными в контексте обработки больших объёмов данных в современных речевых технологиях.

В связи с указанными трудностями исследователи стремятся автоматизировать процесс фонетической транскрипции посредством алгоритмов распознавания речи и разработка инструментов, способных автоматически выполнять разметку аудиоданных на фонетическом уровне, приобретает особую значимость. Такие средства не только ускоряют процессы лингвистического анализа и повышают качество систем распознавания речи, но и способствуют более глубокому пониманию фонетических особенностей языковых данных.

Автоматическая фонетическая транскрипция играет существенную роль не только в создании новых транскрипций, но и в проверке точности уже существующих. Она позволяет обнаруживать расхождения между реальным звучанием и записанной транскрипцией при аннотировании аудиозаписей. Вопрос оценки качества транскрипций также является актуальным: даже сделанные вручную фонетические транскрипции могут содержать

ошибки, поэтому их точность необходимо оценивать перед использованием. Независимо от того, были ли транскрипции получены автоматически или созданы вручную, они служат базой для последующей обработки, включая лингвистический анализ и обучение систем автоматического распознавания речи (ASR). Поскольку транскрипции рассматриваются как отображение или измерение речевого сигнала, важно определить степень их соответствия стандартам качества, надёжности и точности, требуемым от любых форм измерений.

В современном контексте для автоматизации процесса фонетической транскрипции речи существуют два разных подхода: независимая от текста сегментация фонем и выравнивание фонем по речи (или принудительное выравнивание). В настоящее время существует несколько инструментов для принудительного выравнивания [3-5], основанных преимущественно на классической системе HMM (Hidden Markov model – скрытая марковская модель), построенной на базе Kaldi [6] или HTK toolkit [7]. К наиболее распространённым относятся программы Forced Alignment and Vowel Extraction [5], Language, Brain and Behavior Corpus Analysis Tool [8], Montreal Forced Aligner [4] и Munich Automatic Segmentation System [3]. Например, работа инструмента MAUS (Мюнхенская автоматическая сегментация) организована следующим образом: пользователи загружают в онлайн-сервис орфографические транскрипции и аудиофайлы высказываний. Затем инструмент Balloon [9] преобразует текст в фонемы с использованием алгоритма преобразования графемы в фонему и словаря исключений. Далее на основе фонологических правил формируются возможные варианты произношения, в результате чего получается направленный ациклический график. Этот график представляет множество априорных статистически взвешенных гипотетических вариантов произношения высказывания, после чего система ASR, использующая скрытые марковские модели (HMM), определяет наиболее вероятный вариант произношения для каждого слова и устанавливает границы сегментов [3].

Однако несмотря на широкое распространение и полезность систем принудительного фонетического выравнивания, они имеют ряд существенных недостатков. Прежде всего, их эффективность напрямую зависит от наличия точных орфографических транскрипций, которые не всегда доступны или могут содержать ошибки, что может быть особенно критичным для малоресурсных языков или диалектов, где корпуса с орфографическими транскрипциями ограничены или вовсе отсутствуют. Кроме того, такие системы часто не учитывают вариативность спонтанной речи, включая диалектные особенности, редукции, ассимиляции и другие фонетические процессы, не отраженные в орфографии. Это приводит к снижению точности выравнивания и ограничивает возможности анализа реальных языковых данных. Работ по независимой от текста сегментация фонем значительно меньше и, как правило, предложенные инструменты менее практичны [10]. В свете указанных ограничений возникает необходимость в разработке альтернативных методов фонетической разметки, не зависящих от орфографического текста. Создание нейросетевых моделей, способных напрямую извлекать фонетическую информацию из аудиосигнала, представляется перспективным направлением. Такие модели, обученные на больших объемах данных, учитывающих разнообразие фонетических реализаций, позволят получать более точную и гибкую фонетическую разметку, применимую как к распространенным, так и к малоресурсным языкам, а также к различным речевым стилям и условиям записи.

В рамках настоящего исследования мы сосредоточимся на первом подходе, т.е., независимой от текста сегментация фонем. Предполагается, что описываемая архитектура нейросети будет полезна для автоматизации процесса фонетического анализа и проверки корректности существующих транскрипций, а также для создания обширных и точных корпусов речи для последующих исследований и практических приложений.

2. Методика проведения исследования и предварительная обработка данных

Для эффективного обучения моделей автоматического распознавания транскрипционных знаков нами была проведена комплексная работа по сбору и подготовке наборов аудиозаписей. Материалом являются аудиозаписи, собранные в различных условиях и объединённые в единый корпус для целей настоящего исследования. Общий объём данных составляет 4890 записей в формате .wav и соответствующих им аннотаций в формате TextGrid. Корпус сформирован на основе трёх ключевых групп записей:

1. Спонтанная речь в естественных условиях: включает 5 часов спонтанной речи 32 участников, полученной из ранее проведённых исследований [11]. Данная группа представляет фонетические варианты русского языка, зафиксированные в естественной коммуникации. Участники общались в непринуждённой обстановке, что позволит модели анализировать особенности спонтанной речи.
2. Лабораторные записи билингвов с эмоциональной окраской: содержит 12 часов речи 62 участников, собранных в рамках настоящего проекта. Исследование сфокусировано на изучении русской речи билингвов с различной эмоциональной экспрессией. Участники – билингвы, свободно владеющие русским языком; им предлагалось произносить фразы, отражающие различные эмоциональные состояния, в частности гнев и радость.
3. Данные из открытых источников платформы Lingvodoc: включает 2 часа лабораторных записей 12 участников [12]. В этой группе представлены преимущественно трёхкратные повторения стословника Сводеша, выполненные носителями урало-алтайских языков. Записи проводились в контролируемых условиях с целью обеспечения высокого качества и сопоставимости данных.

Все аудиозаписи сопровождены аннотациями в формате TextGrid, где границы звуков были вручную размечены и проверены тремя независимыми аудиторами-лингвистами с использованием программного обеспечения Praat [13]. Записи осуществлялись с применением высококачественного аудиооборудования при частоте дискретизации 44,1 кГц в условиях, минимизирующих влияние посторонних шумов.

При формировании набора данных особое внимание уделялось обеспечению максимального разнообразия по ряду ключевых параметров: полу, возрасту, диалектной принадлежности и эмоциональному состоянию говорящих. Такой методологический подход позволил создать репрезентативную выборку, повышающую обобщающую способность обучаемых моделей и расширяющую их применимость к широкому спектру речевых особенностей и вариативности естественной речи.

Таким образом, сформированный корпус данных предоставляет надёжную основу для обучения и оценки моделей автоматического распознавания транскрипционных знаков, способствуя развитию более точных и универсальных систем обработки речи.

В процессе анализа и предварительной обработки собранных транскрипций была выявлено неоднородность используемых систем транскрипции и символов. Данный фактор создавал значительные сложности для последующего обучения моделей, поскольку использование разнородных транскрипционных систем приводит к несогласованности данных и потенциально снижает эффективность моделей. Для преодоления этой проблемы нами был осуществлен ряд преобразований файлов аннотаций, направленных на унификацию и стандартизацию транскрипционных данных:

- Стандартизация транскрипций: для обеспечения консистентности данных нами был разработан скрипт на языке Python, предназначенный для автоматизации процесса конвертации и обработки транскрипционных данных. Скрипт [14] осуществляет преобразование существующих транскрипционных символов в формат

Международного фонетического алфавита (IPA), что позволяет унифицировать данные и облегчить их последующий анализ.

- Разработка алгоритмов для преобразования файлов с аннотациями: нами был создан подробный словарь соответствий (mapping), где ключами являются исходные символы, а значениями – соответствующие им символы IPA. Если символ интервала присутствовал в словаре, скрипт заменял оригинальный символ на его эквивалент в IPA. Процесс сопровождался проверкой на необходимость изменения (то есть, если исходный и преобразованный символы не совпадают) и ведением статистики о количестве модифицированных интервалов (см. табл. 1).

Табл. 1. Фрагмент словаря соответствий символов не входящих в IPA.
Table 1. A fragment of the dictionary of matches of characters not included in the IPA.

Значение в исходном файле TextGrid	Значение после работы скрипта
'r'	'g'
"\\ng"	"ŋ"
"tt͡ʃ"	"t͡ʃ"

- Обработка нераспознанных и неоднозначных символов: в случаях, когда символ интервала не был найден в словаре соответствий, скрипт проверял его наличие в списке (unassigned_symbols), который содержал символы, не поддающиеся однозначному сопоставлению или отсутствующие в стандарте IPA. Если символ был обнаружен в этом списке, интервал считался проблемным и ему присваивалось значение “unknown”. Данное решение позволило избежать включения неверных или неоднозначных данных в итоговый набор транскрипций, сохраняя при этом возможность дальнейшего анализа этих случаев отдельно (см. табл. 2).

Табл. 2. Фрагмент словаря «unassigned_symbols» неизвестных символов и символов не входящих в IPA.
Table 2. A fragment of the «unassigned_symbols» dictionary for unknown characters and characters not included in the IPA.

Значение в исходном файле TextGrid	Значение после работы скрипта
"u\\:f"	unknown
"x̆͡ʲç"	unknown
"čč"	unknown
"c̆"	unknown
"ɦ͡нᵝ"	unknown
"œ̬"	unknown
"β̬"	unknown
"ú"	unknown

- Исключение «unknown» символов при обучении: все транскрипционные знаки, помеченные как «unknown», были замаскированы или исключены из обучающей выборки. Данное решение было принято для предотвращения внесения шумовых данных и обеспечения чистоты обучающего набора, что способствует повышению производительности и точности моделей распознавания (см. табл. 3).

Далее был сформирован полный набор уникальных фонем, присутствующих в транскрипциях аудиозаписей. Для этого из каждой транскрипции извлекался список фонем, при этом были исключены все неопознанные или неизвестные символы, замененные на «unknown». Полученный набор фонем был объединен и приведен к множеству уникальных значений, что позволило определить полный список фонем, которые необходимо распознавать модели, исключая шумовые и некорректные данные.

На основе извлеченного набора был создан упорядоченный список фонем. Каждой фонеме был присвоен уникальный числовой идентификатор, формируя таким образом словарь

фонем. Дополнительно в словарь был добавлен специальный символ «|», предназначенный для представления пауз или возможных пропущенных в ходе обработки символов. Данный словарь служил для преобразования последовательностей фонем из транскрипций в числовые последовательности, которые были использованы при обучении нейронной сети.

Табл. 3. Фрагмент обработанного набора данных с заменой неизвестных символов.
Table 3. A fragment of the processed dataset with the replacement of unknown characters.

Название файла	Транскрипция
382.wav	j unknown unknown j
72.wav	d r ɤ
120181.wav	s unknown l unknown s ɛ n ø k s h ɛ unknown p unknown y j ɘ s
1953.wav	ʃ d ɐ r unknown
745.wav	p unknown l z unknown q
66.wav	t unknown n unknown d l
355.wav	p ɜ m unknown
2255.wav	e l ɨ o
2241.wav	k unknown t
1748.wav	s v unknown t j unknown s v a t j unknown s v unknown t j œ
2094.wav	k ɤ n d unknown k
221.wav	p y j p y j p y j
547.wav	ø t unknown unknown unknown ɔ
2080.wav	n ø m
631.wav	ɤ l d œ
1210.wav	j e m b ɘ l j e m b œ l j e m b ɘ l
139.wav	ø d ɘ ø ɛ
2862.wav	j a v ɜ p unknown m ɑ l ʌ k ʌ z unknown f t r unknown

Ключевым этапом подготовки данных явилась токенизация, включающая:

- Загрузку и предварительную обработку аудиоданных: каждая аудиозапись загружается и приводится к единой частоте дискретизации – 16 кГц. Это обеспечивает стандартизацию аудиоданных и повышение качества извлеченных признаков.
- Обработку транскрипций: транскрипции разбиваются на отдельные фонемы. Все вхождения слова «unknown» маскируются, чтобы исключить неопознанные символы из дальнейшего анализа.
- Кодирование фонем: каждая фонема из транскрипции преобразуется в соответствующий числовой индекс с помощью ранее созданного словаря фонем. Если фонема по каким-либо причинам отсутствует в словаре, ей присваивается индекс специального символа «|», что позволяет обработать все возможные случаи без возникновения ошибок.

- Извлечение признаков из аудиоданных: используя инициализированный аудиопроцессор, аудиоданные преобразуются в тензоры входных значений, подходящие для подачи на вход нейронной сети. Процессор производит необходимую нормализацию и, при необходимости, выравнивание по длине (padding).
- Формирование меток: закодированные числовые последовательности фонем используются в качестве меток (labels) для обучения модели – это связывает каждую аудиозапись с соответствующей последовательностью фонемных индексов.
- Объединение данных: обработанные аудиоданные и соответствующие им метки объединяются в единый набор данных, готовый для обучения нейронной сети.

Процесс токенизации и предварительной обработки данных был применен ко всему набору данных, таким образом, каждый образец в наборе данных прошел через описанные этапы преобразования, что обеспечивает единообразие и согласованность данных.

3 Описание признаков

В настоящем исследовании в качестве стандартных акустических признаков использовались мел-частотные кепстральные коэффициенты (MFCC) и мел-спектрограммы (Melspec). MFCC представляют собой спектральные характеристики, отражающие энергетическое распределение сигнала по частотам на нелинейной мел-шкале, которая соответствует восприятию частот человеческим слухом.

Процесс извлечения MFCC включает следующие этапы:

1. Преобразование в частотную область: Исходный речевой сигнал разбивается на короткие временные отрезки (фреймы), к которым применяется быстрое преобразование Фурье (FFT) для перехода в частотную область.
2. Преобразование спектра в мел-шкалу: Полученные спектры мощности проходят через банк мел-фильтров, что позволяет акцентировать частоты, значимые для человеческого восприятия.
3. Логарифмирование и обратное преобразование: Логарифм спектра в мел-шкале вычисляется для сжатия динамического диапазона, после чего применяется обратное дискретное косинусное преобразование (IDCT) для получения MFCC.

Для учета динамических характеристик речи были вычислены производные первого и второго порядка от MFCC:

- Δ (дельта) коэффициенты: представляют скорость изменения MFCC во времени, что позволяет моделировать переходные процессы между фонемами.
- ΔΔ (дельта-дельта) коэффициенты: отражают ускорение изменений MFCC, захватывая более сложные динамические аспекты речевого сигнала.

Форманты являются резонансными частотами голосового тракта, возникающими при артикуляции звуков речи. Особенно важны первые две форманты, F1 и F2, для различения гласных звуков:

- F1: соответствует вертикальному положению языка (степени открытости гласного звука).
- F2: связан с горизонтальным положением языка (передний, средний или отодвинутый назад ряд гласного звука).

Извлеченные формантные частоты сравнивались с типичными значениями для различных гласных звуков на основе фонетических исследований [11]. Для каждого гласного вычислялась абсолютная разница между эталонными формантными частотами и измеренными значениями F1 и F2.

Среднеквадратическое значение энергии (RMS) использовалось для оценки средней амплитуды сигнала в каждом фрейме, что характеризует громкость звука.

Все упомянутые признаки объединялись в единый вектор для каждого временного фрейма. В результате каждый фрейм представлен набором характеристик, отражающих его спектральные, динамические, артикуляционные и энергетические свойства.

4 Архитектура нейронной сети

Наша цель заключалась в создании системы, способной с высокой точностью распознавать фонемы в потоке речи, используя современные методы глубокого обучения и обработки сигналов. В последние годы модели на основе глубоких нейронных сетей продемонстрировали значительный прогресс в задачах автоматического распознавания речи [15, 16], а использование трансформерных архитектур улучшило качество обработки последовательностей [17].

В основе модели лежит позиционное кодирование, которое учитывает порядок звуков в речи. Поскольку последовательность элементов имеет решающее значение для понимания смысла высказывания, мы добавили информацию о позиции каждого звука в последовательности. Такой подход соответствует практике применения позиционного кодирования в трансформерных моделях для учета порядка элементов [17], что позволяет модели различать одинаковые звуки в разных контекстах и учитывать синтагматические связи между фонемами, немаловажно, что данный подход ранее был успешно продемонстрирован в работах по обработке естественного языка [18].

При обучении модели мы столкнулись с неравномерным распределением фонем и разной значимостью ошибок в их распознавании. Для решения проблемы мы разработали пользовательскую функцию потерь с весами и маскированием. Подобные методы взвешивания классов также ранее применялись для борьбы с дисбалансом данных в задачах классификации [19, 16]. Разработанная функция позволяет назначать разные веса разным классам фонем, что обеспечивает фокусировку модели на более редких или критически важных звуках. Маскирование используется для игнорирования нерелевантных или отсутствующих данных в последовательности, повышая устойчивость модели к шумам и паузам [20].

На вход модель принимает последовательности акустических признаков, включающих как общие характеристики звука, так и специфические акустические параметры (см. раздел 3).

Для извлечения локальных паттернов в акустическом сигнале используются свёрточные слои (Conv1D), предназначенные для выявления значений частот или изменения амплитуды, характерные для конкретных фонем [21]. Следом за ними идут рекуррентные слои на основе архитектуры "Long short-term memory" (LSTM), которые позволяют модели сохранять и использовать информацию о предыдущих элементах последовательности, что особенно важно для учёта контекста и последовательности звуков в речи [22].

Далее в архитектуре применяются двунаправленные рекуррентные слои (Bidirectional LSTM), позволяя модели учитывать как предыдущий, так и последующий контекст при распознавании каждой фонемы. Следующие далее механизмы внимания (Attention) интегрированы в модель для того, чтобы она могла выделять наиболее значимые части последовательности при предсказании текущей фонемы [23]. В качестве дополнительного слоя внимания мы добавили референсные значения формант гласных фонем. Данные значения представляют собой эталонные частоты первых двух формант (F1 и F2) для каждой гласной, что позволяет модели более точно сопоставлять входные акустические признаки с соответствующими фонемами (см. табл. 4) [24, 11].

Представленные в табл. 4 значения были рассчитаны в диапазоне ±100 Гц для учёта индивидуальных вариаций и особенностей произношения. Предполагалось, что интеграция

референсных значений формант в механизм внимания позволит модели более точно выделять гласные фонемы, даже при наличии акцентов или фонетических искажений [25].

Чтобы предотвратить переобучение и улучшить генерализацию модели, мы применили слои нормализации и регуляризации [26], что особенно важно при работе с аудиозаписями, содержащими различные варианты произнесения.

В выходных слоях модель преобразует внутренние представления в вероятностное распределение по классам фонем. На основе этого распределения она принимает решение о конкретной фонеме в заданной позиции последовательности, обеспечивая точное и последовательное распознавание.

Табл. 4. Референсные значения гласных фонем.
Table 4. Reference values of vowel phonemes.

i	240, 2400
e	390, 2300
ɛ	610, 1900
a	850, 1610
ɑ	750, 940
ɔ	500, 700
o	360, 640
u	250, 595
y	235, 2100
ø	370, 1900
œ	585, 1710
ɶ	820, 1530
ɒ	700, 760
ʌ	600, 1170
ɤ	460, 1310
ɯ	300, 1390
ɨ	300, 1660
ʉ	320, 1390
ɘ	400, 1720
ɵ	430, 1390
ɜ	550, 1640
ɞ	560, 1330

Общая концепция работы модели заключается в том, что она принимает на вход аудиофайл речи, который предварительно преобразуется в последовательность акустических признаков с учётом формантных характеристик. Позиционное кодирование добавляет информацию о расположении каждого звука. Затем, через комбинацию свёрточных и рекуррентных слоёв, модель извлекает как локальные, так и глобальные зависимости в данных. Механизм внимания, обогащённый референсными значениями гласных, позволяет сконцентрироваться на наиболее значимых признаках для текущего предсказания. На выходе модель генерирует последовательность фонем, осуществляя тем самым фонетическую транскрипцию исходного аудиосигнала.

Поскольку в данных могут присутствовать позиции, которые не следует учитывать при вычислении ошибки (например, заполнители или специальные токены), была разработана взвешенная маскированная функция потерь WeightedMaskedLoss, основанная на категориальной перекрестной энтропии и включающая в себя механизмы маскирования и взвешивания классов. Принцип расчета функции потерь:

1. Маскирование незначимых позиций: вычисляется маска mask, представляющая собой тензор, где элементы равны 1, если соответствующая метка верного значения

(y_true) не равна -1 (то есть значима для обучения), и 0 в противном случае. Это позволяет исключить из расчета ошибки те позиции, которые не несут полезной информации.

2. Преобразование меток: исходные метки y_true преобразуются в целочисленный формат y_true_int. Затем с помощью маски формируется новый тензор меток y_true_fixed, где незначимые позиции заменяются на нули. Это необходимо для корректного вычисления перекрестной энтропии.
3. Вычисление базовой ошибки: с помощью функции sparse_categorical_crossentropy рассчитывается базовая ошибка между предсказанными значениями y_pred и преобразованными метками y_true_fixed.
4. Применение весовых коэффициентов классов: для компенсации дисбаланса классов используется вектор весовых коэффициентов weights. Путем создания one-hot представления меток y_true_one_hot и последующего умножения на веса получается тензор class_weights, содержащий веса для каждого примера. Затем ошибка умножается на соответствующие веса классов.
5. Применение маски к ошибке: ошибка умножается на маску mask, чтобы исключить вклад незначимых позиций.
6. Нормализация ошибки: итоговая ошибка рассчитывается как сумма взвешенных ошибок, деленная на сумму элементов маски. Это обеспечивает корректное усреднение ошибки только по значимым позициям.

5 Результаты

Оценка метрик производительности разработанной модели свидетельствует о достижении следующих показателей: Точность (Precision) – 77%, Полнота (Recall) – 65%, F1-мера – 70,50%, и общая Точность классификации (Accuracy) – 65,77%. Для более глубокого понимания классификационных способностей модели был использована матрица несоответствий (см. рис. 1). Матрица несоответствий (confusion matrix) является широко используемым инструментом для оценки классификационных моделей, поскольку сопоставляет фактические классы данных с предсказанными моделью, тем самым выявляя типы и частоты ошибок [27].

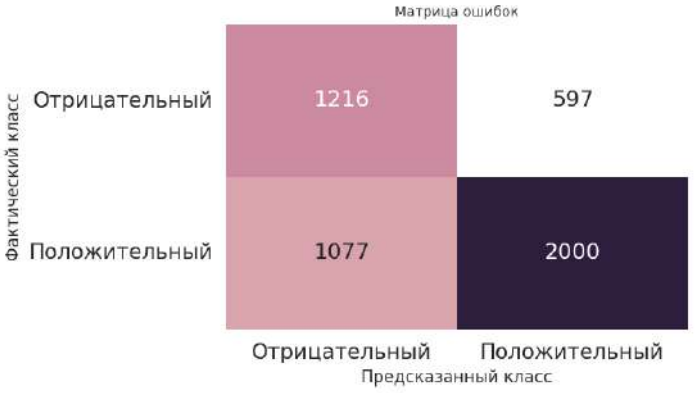


Рис. 1. Матрица несоответствий модели автоматической транскрипции.
Fig. 1. The confusion matrix of the automatic transcription model.

Анализируя матрицу несоответствий, было отмечено, что количество правильно предсказанных положительных экземпляров (истинно положительные, когда класс 1 правильно предсказан как класс 1) составляет 2000, количество правильно предсказанных

отрицательных экземпляров (истинно отрицательные, когда класс 0 правильно предсказан как класс 0) – 1216. Однако имеются значительные ошибки классификации: 1077 случаев, когда положительные экземпляры были неверно предсказаны как отрицательные (ложноотрицательные), и 597 случаев, когда отрицательные экземпляры были неверно предсказаны как положительные (ложноположительные).

Модель демонстрирует более высокую точность для класса 1 (положительный класс) – 77% по сравнению с точностью 65% для класса 0 (отрицательный класс). Одним из потенциальных факторов, влияющих на производительность модели, является несбалансированность частоты уникальных фонем в наборе данных. Анализ распределения фонем выявляет существенные диспропорции в их встречаемости (см. табл. 5):

- Фонемы с высокой частотой: 'l' (2153 вхождения), 't' (1942), 'n' (1936), 'm' (1853), 'k' (1807), 'p' (1673).
- Фонемы с низкой частотой: 'ʧ' (115 вхождений), 'ɵ' (191), 'f' (163), 'ɒ' (166), 'u' (172), 'w' (346).

Табл. 5. Частота встречаемости уникальных фонем в наборе данных.
Table 5. The frequency of occurrence of unique phonemes in the dataset.

m: 1853	v: 997
j: 1687	ʌ: 1050
ɤ: 999	q: 680
d: 1293	z: 407
r: 1449	ʒ: 751
p: 1673	i: 622
l: 2153	e: 1075
ø: 1182	a: 947
k: 1807	w: 346
ɘ: 1008	o: 513
s: 981	ɨ: 1138
n: 1936	ʧ: 115
y: 491	ɵ: 191
ɛ: 1094	b: 241
h: 377	ɯ: 268
ʃ: 1023	f: 163
ɞ: 962	ɑ: 280
ɔ: 1092	ʉ: 192
x: 518	ɒ: 166
œ: 1211	u: 172
ɶ: 740	t: 1942
unknown: 3813 (не учитывается при обучении)	

Данная несбалансированность указывает на то, что некоторые фонемы представлены чрезмерно, тогда как другие – недостаточно. Например, фонема 'l' встречается почти в 19 раз чаще, чем 'ʧ'. Такие диспропорции могут привести к смещённому процессу обучения, при котором модель становится более способной распознавать шаблоны, связанные с частыми фонемами, в то время как ей не хватает данных для обучения на редких фонемах.

Неравномерное распределение фонем может существенно повлиять на способность модели обобщать результаты по всем классам. В задачах классификации модели, обученные на несбалансированных наборах данных, склонны отдавать предпочтение более

многочисленному классу, что приводит к более высоким уровням ошибок для малочисленных классов [28]. В контексте задач классификации фонем это означает, что модель может неэффективно изучать характеристики менее частых фонем, что приводит к увеличению количества ошибок, когда эти фонемы присутствуют во входных данных.

Например, модель может правильно классифицировать слова, содержащие высокочастотные фонемы, такие как 'l', 't' или 'n', благодаря обилию обучающих примеров. Напротив, она может ошибочно классифицировать слова, содержащие низкочастотные фонемы, такие как 'tʃ' или 'f', поскольку она недостаточно изучила их связанные шаблоны. Это может способствовать увеличению количества ложноотрицательных и ложноположительных результатов, как это наблюдается в матрице неточностей.

4 Заключение

В результате анализа был сделан вывод о том, что невысокая точность модели в 65,77% отчасти является следствием несбалансированности частоты фонем в наборе данных. Чрезмерное представление некоторых фонем, таких как 'l', 't' и 'n', и недостаточное представление других, таких как 'tʃ', 'f' и 'u', создают условия обучения, при которых модель не учится одинаково на всех примерах. Чтобы смягчить влияние несбалансированности фонем на производительность модели, можно применять несколько стратегий:

1. Аугментация Данных: Увеличение количества экземпляров, содержащих редкие фонемы, с помощью методов аугментации данных может помочь сбалансировать набор данных и предоставить модели больше примеров для обучения [1].
2. Методы балансировки классов: Применение методов, например, с понижением количества экземпляров большинства (under-sampling) может скорректировать распределение классов в наборе данных, что потенциально улучшит способность модели к обобщению [24].
3. Инженерия Признаков: Внедрение методов взвешивания фонем или внедрения обученных векторов признаков, которые учитывают частоту фонем, может помочь модели уделять больше внимания малочисленным фонемам во время обучения.

Таким образом, устранение несбалансированности частоты фонем является критически важным для повышения производительности модели. Путём реализации стратегий по балансировке набора данных и соответствующей настройке алгоритмов обучения возможно улучшить способность модели точно классифицировать как частые, так и редкие фонемы, тем самым увеличивая общую точность. В целом, полученные результаты подтверждают работоспособность предложенного подхода, однако указывают на необходимость дальнейшего совершенствования модели и метода обучения для достижения более высоких показателей точности и полноты в задаче распознавания речи. В дальнейшем планируется взять больший объем однородного материала на одном языке, а затем на близкородственных языках.

Список литературы / References

[1]. Shorten, C., Khoshgoftaar, T. M. A survey on Image Data Augmentation for Deep Learning // Journal of Big Data, 6(1):60, 2019. Доступно по ссылке: https://www.researchgate.net/publication/334279066_A_survey_on_Image_Data_Augmentation_for_Deep_Learning (Дата обращения 23.01.2025).

[2]. Cucchiarini, C., 1993. Phonetic transcription: a methodological and empirical study. Ph.D. thesis, University of Nijmegen, Nijmegen, The Netherlands. Доступно по ссылке: https://repository.ubn.ru.nl/bitstream/handle/2066/145701/mmubn000001_170795853.pdf (Дата обращения 23.01.2025).

[3]. Kisler T., Schiel F., Sloetjes, H. Signal processing via web services: the use case WebMAUS. // Digital Humanities Conference 2012. 2012. pp. 30-34. Доступно по ссылке:

https://www.researchgate.net/publication/248390251_Signal_processing_via_web_services_the_use_case_WebMAUS (Дата обращения 23.01.2025).
[4]. McAuliffe M., Socolof M., Mihuc S., Wagner M., Sonderegger M., Montreal forced aligner: Trainable text-speech alignment using Kaldi // Proc. Interspeech, vol. 2017. 2017. pp. 498– 502.
[5]. Rosenfelder I., Fruehwald J., Evanini K., Yuan, J. FAVE (forced alignment and vowel extraction) program suite. 2011. Доступно по ссылке: http://fave.ling.upenn.edu (Дата обращения 23.01.2025).
[6]. Povey D., Ghoshal A., Boulianne G., Burget L., Glembek O., Goel N., Hannemann M., Motlíček P., Qian Y., Schwarz P., Silovský J., & Stemmer G., Vesel K. The Kaldi Speech Recognition Toolkit // IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, 2011. Доступно по ссылке: https://www.danielpovey.com/files/2011_asru_kaldi.pdf (Дата обращения 23.01.2025).
[7]. Young S., Evermann G., Kershaw D., Moore G., Odell J., Ollason D., Povey D., Valtchev V., Woodland P., The HTK book // Cambridge university engineering department, vol. 3, no. 175, pp. 12. 2002. Доступно по ссылке: https://www.danielpovey.com/files/htkbook.pdf (Дата обращения 23.01.2025).
[8]. Fromont R., Hay J. LaBB-CAT: an Annotation Store // Proceedings of the Australasian Language Technology Association Workshop 2012, 2012. pp. 113–117. Доступно по ссылке: https://aclanthology.org/U12-1015.pdf (Дата обращения 23.01.2025).
[9]. Uwe R. PermA and Balloon: Tools for string alignment and text processing // paper no. 346. 2012. doi: 10.21437/Interspeech.2012-509 (Дата обращения 23.01.2025).
[10]. Teytaut Y., Roebel A. Phoneme-to-Audio Alignment with Recurrent Neural Networks for Speaking and Singing Voice // Proceedings of Interspeech 2021, International Speech Communication Association, Aug 2021, Brno, Czech Republic. pp.61-65, 10.21437/interspeech.2021-1676. hal-03552964 Доступно по ссылке: https://hal.science/hal-03552964/file/1676anav.pdf (Дата обращения 23.01.2025).
[11]. Гончарова О.В. Артикуляционно-акустические характеристики безударных и ударных гласных на месте орфографического ‘а’ в речи носителей разных фоновариантов русского языка // Филологические науки. Вопросы теории и практики. 2024. Том 17. Выпуск 5. 2024. Volume 17. С. 1661-1668. Доступно по ссылке: https://philology-journal.ru/article/phil20240240/fulltext (Дата обращения 23.01.2025).
[12]. Веб-сайт https://lingvodoc.ispras.ru/dictionaries_all (Дата обращения 23.01.2025).
[13]. Boersma P., Weenink D. PRAAT: Doing phonetics by computer. 2024. Доступно по ссылке: https://www.fon.hum.uva.nl/praat/ (Дата обращения 23.01.2025).
[14]. Веб-сайт https://github.com/brainteaser-ov/textgrid (Дата обращения 23.01.2025).
[15]. Graves, A., Mohamed, A., Hinton, G. Speech recognition with deep recurrent neural networks // International Conference on Acoustics, Speech and Signal Processing. 2013. pp. 6645-6649. Доступно по ссылке: https://arxiv.org/abs/1303.5778 (Дата обращения 23.01.2025).
[16]. Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. r., Jaitly, N., Kingsbury, B. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups // IEEE Signal Processing Magazine, 29(6). 2012. pp. 82-97. Доступно по ссылке: https://www.cs.toronto.edu/~hinton/absps/DNN-2012-proof.pdf (Дата обращения 23.01.2025).
[17]. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Polosukhin, I. Attention is all you need // Advances in Neural Information Processing Systems. 2017. pp. 5998-6008. Доступно по ссылке: https://arxiv.org/abs/1706.03762 (Дата обращения 23.01.2025).
[18]. Devlin, J., Chang, M. W., Lee, K., Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). 2019. pp. 4171–4186. Доступно по ссылке: https://aclanthology.org/N19-1423.pdf (Дата обращения 23.01.2025)
[19]. Cui, Y., Jia, M., Lin, T. Y., Song, Y., Belongie, S. Class-balanced loss based on effective number of samples // Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019. pp. 9268-9277. Доступно по ссылке: https://arxiv.org/abs/1901.05555 (Дата обращения 23.01.2025)
[20]. Park, D. S., Chan, W., Zhang, Y., Chiu, C. C., Zoph, B., Cubuk, E. D., Le, Q. V. SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition // Proc. Interspeech 2019. 2019. pp. 2613-2617. Доступно по ссылке: https://arxiv.org/abs/1904.08779 (Дата обращения 23.01.2025).
[21]. Sainath, T. N., Weiss, R. J., Senior, A., Wilson, K. W., Vinyals, O. Learning the speech front-end with raw waveform CLDNNs // Proc. Interspeech 2015. 2015. pp. 1-5. Доступно по ссылке: https://www.ee.columbia.edu/~ronw/pubs/interspeech2015-waveform_cldnn.pdf (Дата обращения 23.01.2025)

[22]. Graves A., Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures // Neural Networks, Volume 18, Issues 5–6. 2005. pp. 602-610. doi.org/10.1016/j.neunet.2005.06.042 (Дата обращения 23.01.2025).
[23]. Bahdanau, D., Cho, K., Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate // Proc. ICLR 2015. 2015. Доступно по ссылке: https://arxiv.org/abs/1409.0473. (Дата обращения 23.01.2025)
[24]. Chawla, N. V., Bowyer, K. W., Hall, L. O., Kegelmeyer, W. P. SMOTE: Synthetic Minority Over-sampling Technique. // Journal of Artificial Intelligence Research. 16. 2002. pp. 321–357. Доступно по ссылке: http://dx.doi.org/10.1613/jair.953 (Дата обращения 23.01.2025).
[25]. Toshniwal, S., Bahdanau, D., Sagayama, S., Bengio, Y. Multitask learning with low-level auxiliary tasks for encoder-decoder based speech recognition // Proc. Interspeech 2017. 2017. pp. 3532-3536. Доступно по ссылке: https://arxiv.org/pdf/1704.01631(Дата обращения 23.01.2025).
[26]. Ioffe, S., Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift // Proc. ICML 2015. 2015. pp. 448-456. Доступно по ссылке: https://arxiv.org/abs/1502.03167 (Дата обращения 23.01.2025).
[27]. Powers, D. M. W. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation // Journal of Machine Learning Technologies. 2(1). 2011. pp. 37–63. Доступно по ссылке: https://arxiv.org/abs/2010.16061 (Дата обращения 23.01.2025).
[28]. He, H., Garcia, E. A. Learning from Imbalanced Data // IEEE Transactions on Knowledge and Data Engineering. 21 (9). 2009. pp. 1263–1284. doi: 10.1109/TKDE.2008.239 (Дата обращения 23.01.2025).

Информация об авторах / Information about authors

Оксана Владимировна ГОНЧАРОВА – кандидат филологических наук, доцент, руководитель научно-образовательного центра «Интеллектуальный анализ данных» ФГБОУ ВО Пятигорский государственный университет, доцент кафедры русского языка и методики его преподавания ФГАОУ ВО Российский университет дружбы народов имени Патриса Лумумбы, старший научный сотрудник лаборатории Лингвистических платформ НИИ «Институт системного программирования им. В. П. Иванникова РАН» (техническая поддержка научно-исследовательской работы) с 2024 года. Сфера научных интересов: акустическая фонетика, просодия, социолингвистика, обработка естественного языка.

Oksana Vladimirovna GONCHAROVA – Cand. Sci. (Philology), Associate Professor, Head of the Scientific and Educational Center "Intellectual Data Analysis" of the Pyatigorsk State University, Associate Professor of the Department of Russian Language and Teaching Methods at Patrice Lumumba Peoples' Friendship University of Russia, Senior Researcher at the Laboratory of Linguistic Platforms of the Ivannikov Institute for System Programming of the Russian Academy of Sciences (technical support for research work) since 2024. Research interests: acoustic phonetics, prosody, sociolinguistics, natural language processing.

DOI: 10.15514/ISPRAS-2025-37(1)-10

CC BY

Соревнования по формальной верификации VeHa-2024: накопленный в течение двух лет опыт и перспективы

[1] *Д.А. Кондратьев, ORCID: 0000-0002-9387-6735 <apple-66@mail.ru>*
[2] *С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>*
[3] *И.В. Шошмина, ORCID: 0000-0001-5120-0385 <shoshmina_iv@spbstu.ru>*
[4] *А.В. Красненкова, ORCID: 0009-0009-5742-1555 <akrasnenkova@astralinux.ru>*
[5, 6] *К.В. Зиборов, ORCID: 0000-0002-5676-9105 <krl.ziborov@gmail.com>*
[7] *Н.В. Шилов, ORCID: 0000-0001-7515-9647 <shiloviis@mail.ru>*
[1] *Н.О. Гаранина, ORCID: 0000-0001-9734-3808 <garanina@iis.nsk.su>*
[4] *Т.Ю. Черганов, ORCID: 0009-0000-6119-9234 <cherganov@gmail.com>*

[1] *Институт систем информатики им. А.П. Ершова СО РАН,*
Россия, 630090, г. Новосибирск, пр. Академика Лаврентьева, 6.
[2] *АлтГТУ им. И.И. Ползунова,*
Россия, 656038, Алтайский край, г. Барнаул, пр. Ленина, 46.
[3] *Санкт-Петербургский политехнический университет Петра Великого,*
Россия, 195251, г. Санкт-Петербург, ул. Политехническая, 29.
[4] *РусБИТех-Астра,*
Россия, 117105, г. Москва, ул. Варшавское шоссе, д. 26.
[5] *Positive Technologies,*
Россия, 107061, г. Москва, Преображенская площадь, 8.
[6] *Московский государственный университет имени М.В. Ломоносова,*
Россия, 119991, г. Москва, Ленинские горы, д. 1.
[7] *Университет Иннополис, Россия, 420500, г. Иннополис, ул. Университетская, д. 1.*

Аннотация. В нашей статье описан опыт организации второго соревнования по формальной верификации программ среди молодых специалистов и студентов российских вузов. Соревнования проводились в связке с семинаром по семантике, спецификации и верификации программ (PSSV) в Иннополисе в октябре 2024 года. Формат соревнования был близок к формату так называемых хакатонов. Участникам было предложено решить на выбор 5 задач по верификации с использованием заранее определенных инструментов проверки моделей и дедуктивной верификации (Frama-C, Coq, C-lightVer, SPIN, TLC). Мы обсуждаем вопросы организации такого мероприятия, предложенные задачи, результаты решений и обратную связь от участников.

Ключевые слова: формальные методы; соревнования по формальной верификации; проверка моделей; дедуктивная верификация; хакатон.

Для цитирования: Кондратьев Д.А., Старолетов С.М., Шошмина И.В., Красненкова А.В., Зиборов К.В., Шилов Н.В., Гаранина Н.О., Черганов Т.Ю. Соревнования по формальной верификации VeHa-2024: накопленный в течение двух лет опыт и перспективы Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 159-184. DOI: 10.15514/ISPRAS–2025–37(1)–10.

VeHa-2024 Formal Verification Contest: Two Years of Experience and Prospects

[1] *D.A. Kondratyev, ORCID: 0000-0002-9387-6735 <apple-66@mail.ru>*
[2] *S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>*
[3] *I.V. Shoshmina, ORCID: 0000-0001-5120-0385 <shoshmina_iv@spbstu.ru>*
[4] *A.V. Krasnenkova, ORCID: 0009-0009-5742-1555 <akrasnenkova@astralinux.ru>*
[5,6] *K.V. Ziborov, ORCID: 0000-0002-5676-9105 <krl.ziborov@gmail.com>*
[7] *N.V. Shilov, ORCID: 0000-0001-7515-9647 <shiloviis@mail.ru>*
[1] *N.O. Garanina, ORCID: 0000-0001-9734-3808 <garanina@iis.nsk.su>*
[4] *T.Y. Cherganov, ORCID: 0009-0000-6119-9234 <cherganov@gmail.com>*

[1] *Ershov Institute of Informatics Systems, Siberian Branch of the Russian Academy of Sciences, 6, Acad. Lavrentjev pr., Novosibirsk 630090, Russia.*
[2] *Polzunov Altai State Technical University, 46, Lenin Ave., Barnaul, 656038, Russia.*
[3] *Peter the Great St. Petersburg Polytechnic University, 29, Politekhnicheskaya street, St. Petersburg, 195251, Russia.*
[4] *RusBITech-Astra, 26, Varshavskoe Highway, Moscow, 117105, Russia.*
[5] *Positive Technologies, 8, Preobrazhenskaya sq., Moscow, 107061, Russia.*
[6] *Lomonosov Moscow State University, 1, Leninskiye Gory, Moscow, 119991, Russia.*
[7] *Innopolis University, 1, Universitetskaya Str., Innopolis, 420500, Russia.*

Abstract. We present our experience of organizing the second contest in formal program verification for young engineers, researchers, and students from Russia. The contest was held in conjunction with the Workshop on Program Semantics, Specification, and Verification (PSSV) in Innopolis in October 2024. The contest format was close to the format of the so-called hackathons. Participants were asked to select and solve any of 5 verification problems using pre-defined model checking and deductive verification tools (Frama-C, Coq, C-lightVer, SPIN, TLC). We discuss the issues of organizing of the contest, the problems set, lessons learned from solutions submitted by contestants, and the overall feedback from the participants.

Keywords: formal methods; formal verification competitions; model checking; deductive verification; hackathon.

For citation: Kondratyev D.A., Staroletov S.M., Shoshmina I.V., Krasnenkova A.V., Ziborov K.V., Shilov N.V., Garanina N.O., Cherganov T.Y. VeHa-2024 Formal Verification Contest: Two Years of Experience and Prospects. Trudy ISP RAN/Proc. ISP RAS, vol 37, issue. 1, 2025., pp. 159-184. DOI: 10.15514/ISPRAS-2025-37(1)-10.

1. Введение и мотивация проведения соревнования

Основная цель соревнования VeHa-2024 – привлечь студентов Российских вузов к решению задач верификации с помощью формальных методов. Это соревнование продолжает соревнование VeHa-2023 [1], которое проводилось нами впервые. После завершения проведения первого такого соревнования была получена положительная обратная связь, поэтому было решено продолжить организацию соревнований, сделав их ежегодным дополнением к семинару по формальной верификации программ PSSV, в настоящее время проходящем в Университете Иннополис. VeHa означает «Verification Hackathon», то есть состязание по верификации, и проводится по правилам такого рода соревнований – за небольшое ограниченное время в одном месте. Для расширения круга участников в нашем соревновании разрешено и онлайн участие. Отличительной особенностью соревнования 2024

года являлось то, что организаторы стремились предложить задачи индустриальной направленности: верификация функций прав доступа, моделирование процесса исполнения параллельных программ, решение всем известной задачи SAT и верификация протокола консенсуса. Часть из этих задач была предложена российскими компаниями, работающими в сфере безопасности программного обеспечения. Таким образом, дополнительной мотивацией текущего соревнования стало показать возможность применения формальных методов для реальных задач.

Дальнейшее содержание статьи представляет собой обзор структуры соревнования (раздел 2), других схожих соревнований по формальной верификации (раздел 3), сравнение с командным соревнованием по программированию (раздел 4), описание предложенных задач (раздел 5), обзор ошибок в решениях (раздел 6), результаты соревнования (раздел 7), а в заключении делаем некоторые выводы (раздел 8).

2. Структура соревнования

Формат мероприятия в 2024 году сложился в результате беседы во время награждения победителей прошлогоднего соревнования из Москвы между сессиями Открытой конференции ИСП РАН 2023. Участниками и победителями того соревнования были представители Группы Астра (известная по продукту Astra Linux), которые после получения наград предложили в следующем году организовать свою секцию по Frama-C и Coq, которые реально используются в их компании в отделе анализа безопасности и формальной верификации. В итоге к двум имеющимся секциям по проверке моделей и дедуктивной верификации на основе инструмента C-lightVer добавились две секции от группы Астра, также сотрудник Positive Technologies и МГУ им. М. В. Ломоносова предложил секцию по проверке модели протокола консенсуса. Кроме этого, планировалась и секция по выводу типов на основе реализуемого в Иннополисе средства Rzk, но данная секция была отменена в последний момент по личным обстоятельствам организатора. Поскольку секций стало больше, чем две, а задачи усложнились (по сравнению с 2023 годом), было решено подводить итоги по каждой из секций отдельно, что отличается от процедуры 2023 года, когда нужно было обязательно решать задачи из обеих секций.

В 2024 году организаторами соревнования являлись:

- Гаранина Наталья (к.ф.-м.н., с.н.с. Института Систем Информатики, Института Автоматики и Электрометрии СО РАН, доцент НГУ) – ответственная за секцию проверки моделей;
- Зиборов Кирилл (аспирант механико-математического факультета МГУ им. М. В. Ломоносова и инженер Positive Technologies) – ответственный за секцию "Построение и проверка модели протокола консенсуса IBFT";
- Кокорин Артём (Старший специалист по анализу безопасности отдела анализа СЗИ и формальной верификации, группа Астра) – Ответственный за секцию Frama-C;
- Кондратьев Дмитрий (к.ф.-м.н., н.с. Института систем информатики им. А.П. Ершова СО РАН и старший преподаватель Новосибирского Государственного Университета) – ответственный за секцию дедуктивной верификации с помощью системы C-lightVer;
- Красненкова Анастасия (Специалист по анализу безопасности отдела анализа СЗИ и формальной верификации, группа Астра) – ответственная за секцию Frama-C;
- Никольский Денис (Университет Иннополис) – координатор задач;
- Сим Виолетта (Университет Иннополис) – координатор задач (Участник соревнования VeHa-2023);
- Старолетов Сергей (к.ф.-м.н., доцент АлтГТУ, с.н.с. Института Автоматики и Электрометрии СО РАН) – ответственный за секцию проверки моделей и за сайт

соревнований;

- Черганов Тимофей (Старший специалист по анализу безопасности отдела анализа СЗИ и формальной верификации, группа Астра) – ответственный за секцию Coq;
- Шилов Николай (к.ф.-м.н, Университет Иннополис) – организатор родительского семинара PSSV, координатор;
- Шошмина Ирина (к.т.н, доцент кафедры распределенных вычислений СПбПУ) – ответственная за секцию проверки моделей.

Были определены координаторы задач, которые проверяли условия задач на непротиворечивость, а материалы к ним на доступность и воспроизводимость примеров.

Задачи, которые вошли в финальный перечень предложенных на соревнование:

1. Верификация функции проверки прав доступа на Frama-C.
2. Верификация функции проверки прав доступа на Coq.
3. Моделирование вычислительного конвейера графического процессора с поиском оптимальных параметров с помощью проверки моделей.
4. Дедуктивная верификация программы, относящейся к задаче проверки выполнимости булевых формул (SAT).
5. Построение и проверка модели протокола консенсуса IBFT.

Через форму предварительной регистрации было подано более 100 заявок, в аффилиациях участников были указаны следующие университеты:

- НГУ;
- МГТУ им Н.Э.Баумана;
- МФТИ;
- РТУ МИРЭА;
- СПБПУ;
- Университет Иннополис;
- МГУ им. М. В. Ломоносова;
- Университет ИТМО.

Как и в 2023 году, был предложен гибридный формат участия: можно было участвовать как онлайн, так и очно в Иннополисе, с проживанием в гостинице и возможностью посетить сессии и экскурсии совместно с участниками семинара PSSV. Распределение участников по планируемой форме участия показано на рис. 1.

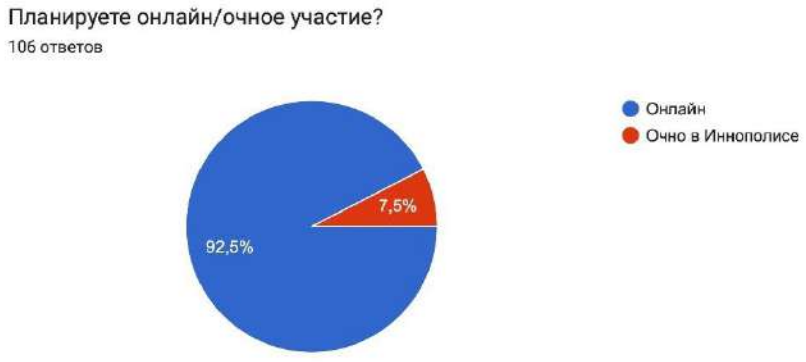


Рис. 1. Распределение голосования за онлайн и очное участие.
Fig. 1. Distribution of votes by online/offline.

В форме регистрации также предлагалось указать планируемую технологию верификации, с небольшим перевесом тут оказалась проверка моделей, см. рис. 2.

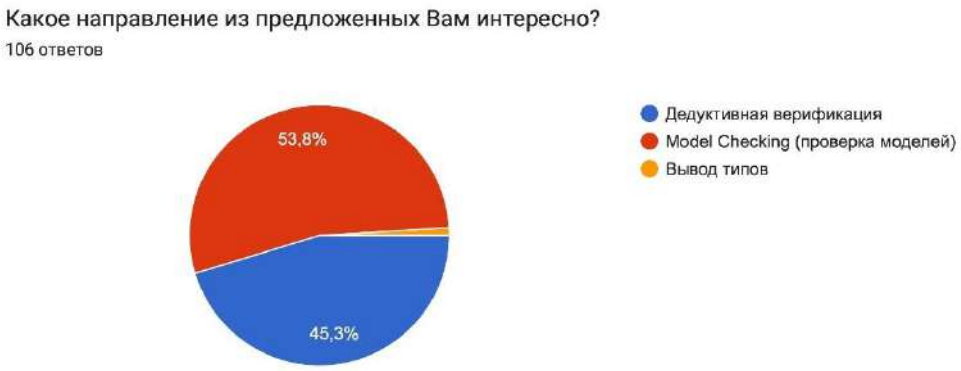


Рис. 2. Распределение предпочтений по технологиям верификации.
Fig. 2. Distribution of preferences by verification technologies.

По предварительной регистрации на задачи небольшой перевес был отдан задаче проверки моделей для конвейера GPU (рис. 3).

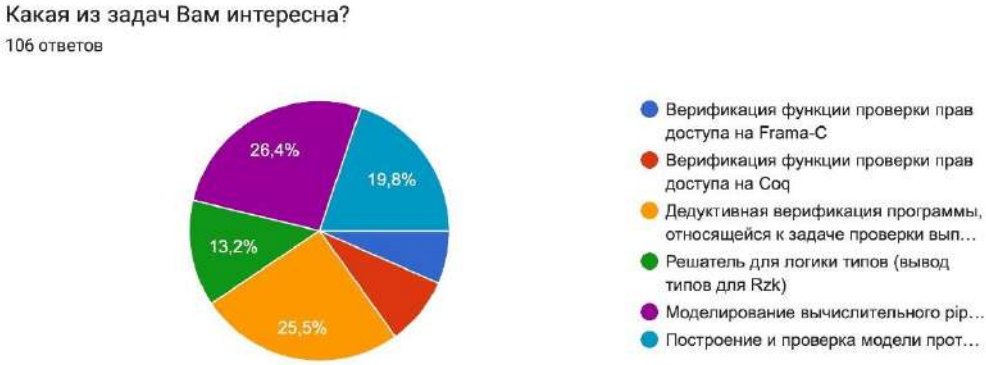


Рис. 3. Распределение выбора по предложенным задачам.
Fig. 3. Distribution of choices by proposed tasks.

Для агрегации всех материалов, как и в прошлом году, мы разработали сайт соревнований на платформе Google Sites. На сайте были размещены материалы по задачам, включая образ виртуальной машины для задач 1 и 2, предварительные описания и полные описания задач соревнования (открыты при старте соревнований), даны ссылки на Telegram-группу для быстрых обсуждений, Skype-группу для обучающих уроков и консультаций, а также GitHub-репозиторий для последующей загрузки решений с использованием технологии «pull-request» (с созданием там своей папки в формате solution_x_name, где x – номер задачи, а name – название команды).

Основное взаимодействие с участниками (объявления, ответы на вопросы, уведомления о появлении новых материалах на сайте) осуществлялось через мессенджер Telegram.

Нами было установлено следующее расписание соревнований:

- 13 октября 2024 (за неделю до старта соревнования) – проведение онлайн-уроков по двум задачам в Skype.
- 10:00 18 октября – публикация полных текстов заданий на сайте.
- Во время соревнований проводились онлайн и очные консультации.
- 23:59 21 октября – завершение загрузки решений.
- 26 октября – встреча оргкомитета соревнования, подведение результатов и решения по грамотам.
- 27 октября – публикация информации о победителях на сайте соревнования.

3. Обзор соревнований, курсов по формальной верификации и сравнение с описанным форматом соревнования

Относительно подробный обзор соревнований по формальной верификации описан в статье о VeHa-2023 [1]. Но, если тот обзор [1] более ориентирован на историю соревнований по формальной верификации, чем на сравнение этих соревнований с мероприятиями серии VeHa, то в данном обзоре рассмотрим более подробно именно такое сравнение вместо исторических аспектов.

Наиболее родственными с серией соревнований VeHa в целом являются мероприятия серии TOOLympics [2-4]. Пока что соревнования TOOLympics состоялись два раза, в 2019 году [2-3] и в 2023 году [4]. Главная схожесть VeHa и TOOLympics состоит в том, что каждое из этих мероприятий объединяет в себе несколько соревнований, покрывающих оба основных направления формальной верификации программ, и дедуктивную верификацию, и проверку моделей. Например, в годы проведения TOOLympics под эгидой данного мероприятия проходят и соревнования по дедуктивной верификации программ серии VerifyThis [2-7], и соревнования по проверке моделей серии RERS [2-4, 8-9]. И, если соревнование VeHa-2023 отличалось от мероприятий серии TOOLympics тем, что итоги VeHa-2023 подводились для всего соревнования суммарно по всем трекам, то уже VeHa-2024 не отличается в этом плане от мероприятий серии TOOLympics, так как итоги VeHa-2024 подводились для каждого трека по-отдельности. Основное отличие VeHa от TOOLympics состоит в том, что на мероприятиях TOOLympics участники могут выбирать любые программные системы для решения задач по верификации, тогда как на соревнованиях VeHa участникам даются указания, какие именно программные системы использовать для решения задач. Таким образом, на мероприятиях TOOLympics важную роль играют не только компетенции участников, но и возможности выбранных участниками программных систем. Это отражено в названии мероприятий TOOLympics. А на соревнованиях VeHa проверяются именно компетенции участников, что ближе к изначальной идее хакатона, отраженной в названии серии VeHa (Verification Hackaton). Другое важное отличие состоит в том, что в рамках TOOLympics проходят соревнования не только в области дедуктивной верификации и проверки моделей, но и соревнования в некоторых других областях формальных методов в программировании, например, соревнование Termination Competition (termCOMP) [2-4, 10] по проверке выполнения свойства завершаемости исполнения программ и систем переписывания термов.

Также родственным в целом для серии VeHa является серии мероприятий SpecifyThis [11-12]. Мероприятия SpecifyThis пока что состоялись два раза, в 2022 году [11], и в 2024 году [12]. Главное отличие SpecifyThis от соревнований VeHa состоит в том, что SpecifyThis не является соревнованием. Мероприятие SpecifyThis больше похоже на рабочий семинар, участники которого обсуждают проблематику задания функциональных спецификаций, относительно которых проверяется корректность программы в ходе дедуктивной верификации, и спецификаций, написанных на языке временной логики, относительно которых проверяется корректность программных систем в ходе проверки моделей.

Обсуждение на мероприятии конкретных примеров задания спецификаций, а также название мероприятия, созвучное названию соревнования VerifyThis, демонстрируют потенциал трансформации в будущем мероприятия SpecifyThis в соревнование по заданию спецификаций программ. Главная схожесть SpecifyThis и серии соревнований VeHa состоит в том, что на большинстве треков VeHa важную роль играет задание функциональных или темпоральных спецификаций программных систем. Таким образом, проблематика задания спецификаций играет важную роль на серии соревнований VeHa в целом.

Кроме того, родственной по отношению к серии соревнований VeHa деятельностью является составление наборов задач, позволяющих проверять возможности различных подходов к формальной верификации программ. Отметим, что такие наборы составляются как отдельно от соревнований [13-16], так и по итогам соревнований по формальной верификации. Например, такой набор задач составляется по итогам серии соревнований по проверке моделей RERS [9, 17]. Особенно отметим набор задач, который составляется по итогам серии соревнований по дедуктивной верификации VerifyThis [5, 18]. Данный набор задач и их решений структурирован по годам проведения соревнования и доступен на отдельной веб-странице [19]. Условия задач прошлогоднего соревнования VeHa-2023 доступны на сайте соревнования [20], а их решения участниками доступны в репозитории соревнования [21]. Аналогично условия задач соревнования VeHa-2024 доступны на сайте соревнования [22], а их решения участниками доступны в репозитории соревнования [23].

И кроме того, важно отметить связь соревнований VeHa с такой деятельностью, как обучение формальной верификации программ. В первую очередь такая связь состоит в активном участии на соревнованиях серии VeHa студентов, проходящих обучение формальным методам в программировании. Например, многие победители и призеры соревнований VeHa из НГУ обучены/обучаются на профиле "Формальные методы анализа программ и систем", где основные дисциплины состоят в обучении различным методам формальной верификации программ [24]. Также отметим успешное выступление на соревновании VeHa-2023 студентов из Университета Иннополиса и соответствующий курс по формальной верификации программ [25]. Участие на соревнованиях позволяет студентам развить полученные на курсах навыки по формальной верификации и лучше понять практическую применимость формальных методов в программировании. Эти результаты соревнования особенно важны, поэтому, чтобы облегчить достижение таких результатов, появилась актуальная задача по встраиванию соревнований VeHa в существующие курсы по формальной верификации программ. Во вторую очередь связь соревнований VeHa с обучением формальным методам состоит в возможности проверить, могут ли студенты успешно применять разрабатываемые организаторами соревнований системы формальной верификации программ. Отметим, что применение систем формальной верификации программ на практике требует привлечения высококвалифицированных специалистов. Разработчики практически всех систем формальной верификации стремятся снизить квалификационные требования к пользователям таких систем. Для этого в таких системах реализуют методы упрощения и автоматизации формальной верификации. Такие подходы, естественно, не могут обеспечить полную автоматизацию формальной верификации, однако важно проверить, могут ли такие методы настолько снизить требования к компетенциям пользователей систем верификации, чтобы такие системы могли применять обучающиеся формальным методам студенты. Также важно получить обратную связь от студентов, какую именно функциональность рекомендуется реализовать в тестируемых таким образом системах, чтобы упростить применение таких систем. Особенно это актуально для системы дедуктивной верификации C-lightVer [26-29], которая разрабатывается в ИСИ СО РАН, одним из организаторов VeHa. Можно провести аналогию с предложениями студентов Иннополиса об улучшениях системы дедуктивной верификации AutoProof по итогам обучения применению данной системы на курсе по формальной верификации [30].

Далее рассмотрим сравнение отдельных треков соревнования VeHa-2024 с родственными соревнованиями по формальной верификации программ.

Треки соревнования VeHa-24 “Верификация функции проверки прав доступа на Frama-C” и “Дедуктивная верификация программы, относящейся к задаче проверки выполнимости булевых формул (SAT)” схожи с соревнованиями серий VSComp [31-32], VerifyThis [5-7, 33-35] и с долговременными соревнованиями VerifyThis [36-38]. Это известные соревнования по дедуктивной верификации программ, проведение которых было во многом вдохновлено проектом Verified Software Initiative по масштабному применению формальной верификации в индустриальном программировании [39-40]. Главным отличием связанных с дедуктивной верификацией треков VeHa от данных соревнований является возможность выбора участниками этих соревнований любых программных инструментов для дедуктивной верификации, что делает результат этих соревнований зависимым не только от компетенций участников, но и от возможностей выбранных программных средств [18, 41]. Отдельно отметим долговременные соревнования VerifyThis. Если обычные соревнования серии VerifyThis ограничены по времени днями проведения научных конференций, в рамках которых проводятся соревнования, то долговременные соревнования проводятся на значительных промежутках времени между научными конференциями. Такая длительность позволяет использовать масштабные задачи по индустриальной верификации, что является еще одним важным отличием от связанных с дедуктивной верификацией треков соревнования VeHa.

Трек соревнования VeHa-2024 “Верификация функции проверки прав доступа на Coq” схож не только с соревнованиями по дедуктивной верификации программ VSComp [31-32], VerifyThis [5-7, 33-35] и долговременными VerifyThis [36-38], но и с соревнованием по интерактивному доказательству теорем Proof Ground [42]. Данный трек соревнования VeHa-2024 посвящен такому этапу дедуктивной верификации, как проверка истинности в системе доказательства (на данном треке система Coq [43]) условий корректности программ. Умение успешно справляться с данным этапом дедуктивной верификации играет важную роль на соревнованиях серий VSComp, VerifyThis и долговременных соревнованиях VerifyThis. Отличие от Proof Ground состоит в том, что доказываемые на соревновании Proof Ground теоремы могут быть из самых разных предметных областей (не только из области условий корректности программных систем).

Треки соревнования VeHa-24 по моделированию вычислительного конвейера графического процессора с поиском оптимальных параметров с помощью проверки моделей и по построению и проверки модели протокола консенсуса IBFT схожи с соревнованиями по проверке моделей серии RERS [8-9, 17, 44-45]. Отметим, что в некоторых задачах соревнований серии RERS, также как и на треках VeHa-2024, для задания верифицируемых свойств в терминах логики LTL применяется язык Promela [17]. Однако на соревнованиях RERS участникам предоставляются уже заданные свойства, которые необходимо верифицировать, тогда как на треках соревнования VeHa участникам необходимо самим задавать верифицируемые свойства и модель, исходя из описания задач на естественном языке.

Из обзора родственных мероприятий можно сделать следующие выводы в отношении соревнований VeHa:

1. Полезно расширять соревнования VeHa новыми треками. В качестве таких треков может быть рассмотрено аналогичное termCOMP [10, 46-48] соревнование по доказательству завершаемости программ. Также отметим, что на соревновании VeHa-2024 планировался трек по автоматическому определению типа выражения в модельном языке с помощью специальной логики типов, однако, по независящим от организаторов причинам, данный трек в этом году был отменен. Такой трек планируется проводить, начиная со следующего соревнования серии VeHa.

2. Полезно накапливать наборы задач и их решений по итогам соревнований VeHa в более структурированном виде.
3. Актуальной задачей является встраивание соревнований серии VeHa в учебные курсы по формальной верификации программ.
4. Для развития навыков участников VeHa по решению масштабных задач по индустриальной верификации можно проводить долговременные соревнования по формальной верификации программ в промежутках между обычными соревнованиями серии VeHa.

4. Сравнение соревнований VeHa и ICPC

Мы уже писали в разделе 3, что родственными с VeHa являются мероприятия серии TOOLympics [2-4], соревнования по дедуктивной верификации программ серии VerifyThis [5-7, 33-35], соревнования по проверке моделей серии RERS [8-9, 17, 44-45] и мероприятия серии SpecifyThis [11-12].

Но есть еще одни студенческие соревнования по программированию, популярность которых среди *десятков тысяч студентов* по всему миру заставляет сравнить эти соревнования с VeHa: речь идет о Международной студенческой олимпиаде по программированию, называемой также Студенческим командным чемпионатом мира по программированию (The International Collegiate Programming Contest, ICPC, до 2017 – ACM ICPC) [49-50]. Первый финал ACM ICPC был проведён в рамках ежегодной конференции ACM по информатике, 5-ой ACM Computer Science Conference (CSC'77) [51] (то есть, как соревнования VeHa проходят в аффилиации с семинаром PSSV [52]).

ICPC – соревнования между университетскими командами из трёх студентов (допускаются аспиранты первого года обучения, но не старше 24 лет), они проходят в несколько туров/этапов: отдельных университетов, региональных и финала. Каждый этап проходит следующим образом: каждой команде выдаётся один компьютер на пять часов, всем командам – один набор из 8-12 задач (условия которых написаны в свободной форме на английском языке).

Команды пишут решения на языках программирования Pascal, C, C++, Java, Python или Kotlin, и отправляют решения на *тестирующий* сервер. Корректность решений, поступивших на тестирующий сервер, проверяется *на большом количестве различных входных тестов*, подготовленных жюри, но неизвестных участникам. Решение считается корректным, если программа прошла все тесты с правильным результатом за специфицированное для каждого теста время, использовав специфицированный для каждого теста объем памяти.

Нам представляется, что ни тестирование, как метод проверки корректности решений на ICPC, ни экспертный метод разработки самих тестов для проверки корректности решений, ни неверифицированные «образцовые» решения, предлагаемые жюри после каждого тура ICPC, нельзя считать современными методами оценки корректности программ и квалификации участников соревнований. Мы считаем, что интеграция формальной верификации в студенческие соревнования по программированию (включая ICPC) является требованием времени для повышения качества оценки решений и повышения уровня образования участников. Теперь вопрос о проведении такого эксперимента на студенческих соревнованиях по программированию.

5. Обзор предложенных задач

Полные условия всех задач можно найти на сайте соревнования VeHa-2024 [22].

5.1 Задача 1. Верификация функции проверки прав доступа на Frama-C

В качестве условий задачи даны спецификации к исходному коду на языке Си из фрагментов кода Linux Kernel в формате ACSL [53], см. листинг 1.

Функция `compute_mask()` вычисляет маску (с битами MAY_WRITE и MAY_READ), которая в дальнейшем используется для принятия решения о разрешении доступа к заданному файлу. Доступы в маске означают наличие прав соответствующего доступа.

Функция `check_permission()` проверяет доступ к заданному файлу с учетом прав пользователя на чтение/запись (см. листинг 2).

```
static int compute_mask(struct file *file, unsigned int cmd)
{
    struct inode *inode = file_inode(file);
    int mask = 0;
    int i = 0;
    // Количество событий в массиве event_numbers
    unsigned long size_array = (sizeof (event_numbers) /
                                sizeof((event_numbers)[0]));
    // Проверяем, является ли файл публичным.
    // Если да, то доступ разрешен сразу
    if (inode->i_flags & SHIFT)
        return 0;

    if (cmd > IMPORTANT) {
        return MAY_WRITE;
    }
    if (cmd < EXOTIC) {
        return MAY_READ;
    }
    // Нужно верифицировать цикл внутри функции
    for (i = 0; i < size_array; ++i)
        if (cmd == event_numbers[i][0]) {
            mask = event_numbers[i][1];
            break;
        }
    if (!mask)
        mask = MAY_READ | MAY_WRITE;
   return mask;
}
```

Листинг 1. Функция “compute_mask”.
Listing 1. “compute_mask” function.

Функция `check_permission()` возвращает 3 значения:

- 0 – доступ к файлу разрешен;
- -13 – доступ к файлу запрещен (макрос NO_PERM);
- -1 – недостаточно высокий уровень целостности пользователя, ошибка доступа (макрос NO_ILEV).

В код одной из функций организаторами была намеренно внесена ошибка. Участникам было необходимо найти её при помощи инструмента Frama-C, не изменяя уже написанные спецификации к коду; а также исправить код (обосновав своё решение) и полностью

верифицировать получившиеся функции (все цели – как сгенерированные Frama-C, так и созданные верификатором – должны быть доказаны).

Для подготовки к решению задания участникам было предложено небольшое пособие по Frama-C и даны ссылки на известные и хорошо зарекомендовавшие себя учебники по данному инструменту верификации.

```
static int check_permission (struct file *file, unsigned int cmd)
{
    const PDPL_T *sl = getCurrentLabel();
    // Вычисляем уровень целостности пользователя
    unsigned int ilev = slabel_ilev(sl);
    // Если пользователь не максимального уровня целостности,
    // то возвращаем ошибку доступа
    if (ilev != max_ilev){
        return -NO_ILEV;
    }
    // Вычисляем маску
    int mask_final = compute_mask(file, cmd);
    // Текущий процесс -- суперпользователь, и у пользователя
    // есть право на запись в файл -- доступ разрешен
    if (current->process->fsuid == 0)
        if ((mask_final & MAY_WRITE)){
            return 0;
        }
    // Текущий процесс -- не суперпользователь, и у пользователя
    // есть право на чтение из файла -- доступ разрешён
    if (!(current->process->fsuid == 0))
        if ((mask_final & MAY_READ)){
            return 0;
        }

    // В других случаях доступ запрещен
    return -NO_PERM;
}
```

Листинг 2. Функция “check_permission”.
Listing 2. “check_permission” function.

5.2 Задача 2. Верификация функции проверки прав доступа на Coq

Задача заключается в формальной верификации одной из функций модуля безопасности ядра Linux [54]. Поведение системы можно описать переходами между её состояниями. Состояние системы из задачи (ядра Linux):

- множество субъектов (например, процессов);
- множество объектов (например, файлов);
- матрица доступов субъектов к объектам;
- метки целостности субъектов;
- метки целостности объектов.

Переход из одного состояния в другое происходит при наступлении события. В задаче рассматривается событие получения доступа субъекта к объекту. Если в запросе есть доступ на запись, то метка целостности объекта не должна превышать метку целостности субъекта.

Функция vsm_inode_permission из модуля безопасности ядра отвечает за проверку метки целостности процесса при обращении к индексному дескриптору (см. листинг 3). Она принимает два аргумента: указатель на индексный дескриптор, запрашиваемые параметры доступа. Функция возвращает 0, если процессу разрешен доступ к индексному дескриптору.

```
#define MAY_EXEC    0x00000001
#define MAY_WRITE   0x00000002
#define MAY_READ    0x00000004
#define EACCES      13

int vsm_inode_permission(struct inode *inode, int mask)
{
        mask &= MAY_WRITE;

        if (!mask)
                return 0;

        const struct security *isec = inode_security(inode);
        const struct security *tsec = cred_security(current_cred());

        if (tsec->ilev >= isec->ilev)
                return 0;

        return -EACCES;
}
```

Листинг 3. Функция "vsm_inode_permission".
Listing 3. "vsm_inode_permission" function.

Участникам были предложены следующие задания:

- Написать спецификацию этой функции.
- Метки целостности функциональной спецификации обладают типом Z. Метки целостности формальной модели системы – элементы решетки. Нужно доказать, что тип Z является решеткой.
- Чтобы показать корректность функциональной спецификации vsm_inode_permission, нужно доказать, что функция inode_permission возвращает 0 тогда и только тогда, когда функция формальной модели checkRight разрешает доступ субъекта к объекту.
- Чтобы убедиться в корректности формальной модели, нужно доказать, что любой переход системы из одного состояния в другое сохраняет следующее свойство: у субъекта есть доступ на запись к объекту только в том случае, если метка целостности объекта не превышает метку целостности субъекта.

Для подготовки к выполнению задачи участникам было предложено краткое введение в Coq.

5.3 Задача 3. Моделирование вычислительного конвейера графического процессора с поиском оптимальных параметров с помощью проверки моделей

Данная задача является переработкой задачи с прошлогоднего соревнования. В задаче предлагается моделировать процесс исполнения вычислительных инструкций видеокартой с графическим процессором (такой процесс называется конвейером или пайплайном). Для

поиска оптимальных параметров (задачи автонастройки) используется метод проверки модели. Верификатор находит контрпримеры, что позволяет определить оптимальные параметры (согласно статьям авторов [55-56]). В 2023 году к этой задаче приступила лишь одна команда, полученное от нее решение было неполным. В этом году задача коллективом авторов была продумана глубже: выделены различные уровни решения, предложены в качестве начальных авторские модели конвейера.

Для решения задач, требующих интенсивных расчетов, которые хорошо распараллеливаются (вроде операций с матрицами, расчет хэш функций для майнинга, приложений ИИ), применяются графические процессоры (GPU). Все эти задачи используют особенности архитектуры графических процессоров, называемой SIMT (single instruction-multiple thread – одна инструкция – множество потоков). Эта архитектура позволяет эффективно решать задачи с одновременным выполнением однотипных операций над разными данными на большом количестве вычислителей. Для программирования задач под архитектуры GPU используются технологии OpenCL (открытая) или CUDA (закрытая от Nvidia), при этом код, исполняемый на видеокарте, пишется на C-подобном языке и называется “ядром”. Этот код впоследствии транслируется в ассемблерный код (PTX для Nvidia), а потом в его бинарное представление для исполнения на видеокарте конкретной архитектуры.

Работа программы, оптимальной по потреблению GPU-ресурсов (времени, памяти, энергии и т.п.), зависит от множества параметров, подбор которых затруднен. Задача подбора таких параметров называется задачей (авто)настройки параллельной программы. Как было показано в работах авторов [55-56], задачу автонастройки возможно решить с помощью метода проверки модели. Для этого необходимо описать модель исполнения настраиваемой программы на выбранной архитектуре GPU. Модель исполнения должна быть описана на входном языке верификатора с учётом настраиваемых параметров. В работах [55-56] авторами была разработана абстрактная модель OpenCL программы, реализована автонастройка параметров оптимизации для программы (WG – размер рабочей группы в программе на OpenCL, TS – размер массива данных, обрабатываемый потоком), написанной на OpenCL, с помощью метода проверки модели.

В предложенных участникам на ознакомление публикациях также проведено моделирование конвейера абстрактного GPU для параллельной программы поиска минимума в большом массиве. Для этого на языке Promela, входном языке верификатора SPIN, в отдельных процессах моделируются модули, соответствующие хосту, устройствам, узлам (т.е. элементы вычислительного конвейера) и программным элементам (отвечающим за модули программы). Процессы взаимодействуют друг с другом с помощью каналов, реализуя таким образом последовательность работы GPU конвейера при решении задачи. В качестве подбираемых параметров моделируется время для доступа к локальной и глобальной памяти, а также используются размеры рабочей группы и разбиения данных. Решение задачи автонастройки с помощью метода проверки модели состоит в том, что делается запрос верификатору на невозможность достижимости конечного состояния с заданными параметрами. Когда верификатор возвращает контрпример, нарушающий запрос, то в этом контрпримере содержатся оптимальные значения параметров.

Для детального моделирования GPU, более приближенного к реальности (с учетом представления программы в виде PTX инструкций, кэша таких инструкций, пулов потоков исполнения (warp, единица планирования планировщика GPU), сборщика операндов, дивергенции ветвей исполнения и т.д.), участникам предлагалось воспользоваться наработками проекта GPGU Sim [57], в котором на основе анализа открытых патентов Nvidia исследователи воссоздали последовательность работы GPU (исполнительный конвейер), который и предлагалось реализовать участникам.

В итоге, на соревнование были вынесены задачи на разных уровнях, от простой модификации решения авторов для другой задачи до детальной реализации конвейера:

- Первый уровень. Решить задачу поиска оптимальных параметров решения задачи суммы четных элементов в большом массиве данных, путем модификации авторского решения и абстрактной Promela-модели для OpenCL программы.
- Второй уровень. Дополнительно к уровню 1 модифицировать решение задачи настройки для OpenCL, добавив работу с потоками исполнения.
- Третий уровень. Реализовать уровень 2 для приложения на PTX, превращая исходную программу в последовательность инструкций, программа на языке Promela должна моделировать работу по выбору и исполнению этих инструкций.
- Четвертый уровень. Дополнительно к уровню 3 реализовать детально исполнительный конвейер для подготовки данных и потоков исполнения.

Поскольку задача предполагает большое число технологий для ознакомления, по ней была проведена обучающая лекция, в которой были рассмотрены следующие темы.

- Архитектура SIMT.
- Архитектуры современных GPU (Nvidia (Pascal, Ampere, Ada), AMD (Polaris)). Показана структура мультипроцессора внутри, исходя из архитектурных документов [58-59]).
- Языки программирования для GPU (OpenCL и CUDA).
- Массивный параллелизм в OpenCL-программах.
- Программирование с делением потоков на локальные и глобальные группы.
- Задачи автонастройки. Автонастройка OpenCL программ.
- Задачи поиска минимума на OpenCL.
- Проверка моделей. Абстрактная модель OpenCL программы.
- Моделирование сущностей абстрактного GPU при работе OpenCL программы на примере задачи поиска минимума.
- Контрпримеры при проверке моделей. Поиск оптимальных параметров при помощи контрпримеров.
- Проект GPGU Sim и основные сущности из конвейера GPU Nvidia.

5.4 Задача 4. Дедуктивная верификация программы, относящейся к задаче проверки выполнимости булевых формул (SAT)

Задача заключается в задании такого инварианта цикла для программы, решающей важную часть задачи 2-SAT, который позволяет успешно верифицировать данную программу в системе C-lightVer. Организатором в качестве задачи было выбрано задание инварианта цикла, так как это одна из главных проблем дедуктивной верификации программ [60-61]. Задача 2-SAT [62-63] была выбрана в качестве примера потому, что в последнее время активно развиваются исследования по формальной верификации программных систем для доказательства теорем [64], в том числе таких специализированных систем, как SAT-решатели и SMT-решатели [65-67]. В качестве системы верификации C-lightVer [26-29] была выбрана потому, что в системе C-lightVer применяется система доказательства теорем ACL2 [68], которая может автоматически применять уже доказанные теоремы в качестве лемм для доказательства других теорем, что упрощает доказательство условий корректности программ.

Участникам соревнования предоставлялись пособие по дедуктивной верификации, содержащее пример задания инварианта цикла, система дедуктивной верификации программ C-lightVer, условие задачи и верифицируемая программа с теорией предметной области.

Для понимания предназначения верифицируемой программы участникам нужно было изучить условие задачи, где относительно подробно излагается проблема выполнимости булевых формул в форме 2-КНФ, то есть задача 2-SAT. В условии задачи описано, как формула в форме 2-КНФ может быть представлена в виде конъюнкции импликаций и как можно представить такую конъюнкцию импликаций в виде графа импликаций. Далее в условии задачи определено важное для задачи 2-SAT свойство, которое вводится для некой переменной формулы x: наличие в графе импликаций пути из вершины x в вершину ¬x и одновременно наличие пути из вершины ¬x в вершину x. Затем в условии задачи описано, почему, если существует переменная формулы, для которой выполнено важное для задачи 2-SAT свойство, то исходная формула невыполнима [63]. Потом в условии задачи объяснено, почему, если для всех переменных формулы не выполнено важное для задачи 2-SAT свойство, то исходная формула выполнима [63]. Участникам соревнования нужно было верифицировать программу, которая по уже построенному транзитивному замыканию графа импликаций проверяет для каждой переменной формулы выполнение важного для задачи 2-SAT свойства. Итак, рассмотрим верифицируемую функцию twosat_solver (см. листинг 4).

```
/* Предусловие */
int twosat_solver(int variable_count,
                  int implication_graph_transitive_closure[]){
    int x = 0;
    int satisfiable = 1;
    /* Инвариант цикла */
    while (x < variable_count && satisfiable == 1){
        if (implication_graph_transitive_closure
                 [x + variable_count + x*2*variable_count] == 1 &&
            implication_graph_transitive_closure
                 [x * 2 * variable_count+x+variable_count
                    * 2 * variable_count]
            == 1) {satisfiable = 0;}
        else {x++;}}
    return satisfiable;}
/* Постусловие */
```

Листинг 4. Функция “twosat_solver”.
Listing 4. “twosat_solver” function.

Данная программа выполняет в цикле поиск переменной, для которой не выполняется важное для задачи 2-SAT условие. Если такая переменная найдена, то функция twosat_solver возвращает значение 0, означающее, что формула невыполнима. Иначе функция twosat_solver возвращает значение 1, означающее, что формула выполнима.

Отметим, что транзитивное замыкание матрицы смежности графа импликаций хранится не в двумерном, а в одномерном массиве, что довольно естественно для программ на языке C. Если интерпретировать транзитивное замыкание матрицы смежности графа импликаций как двумерный массив, то размерность такого массива будет 2 * variable_count на 2 * variable_count, где variable_count является количеством переменных в формуле. Индексы строк такого двумерного массива в диапазоне [0 ... variable_count-1] соответствуют переменным формулы x_0, ..., $x_{variable\_count-1}$ без отрицаний, а индексы в диапазоне [variable_count ... 2*variable_count-1] соответствуют переменным формулы $\neg x_0$, ..., $\neg x_{variable\_count-1}$ с отрицаниями. Аналогично индексы столбцов такого двумерного массива в диапазоне [0 ... variable_count-1] соответствуют переменным формулы x_0, ..., $x_{variable\_count-1}$ без отрицаний, а индексы в диапазоне [variable_count ... 2 * variable_count -1] соответствуют

переменным формулы $\neg x_0$, ..., $\neg x_{variable\_count-1}$ с отрицаниями. При переходе к одномерному массиву нужно использовать умножение индекса строки на длину строки и прибавить смещение на индекс столбца. Итого, проверка наличия пути из x в ¬x и проверка наличия пути из ¬x в x осуществляется с помощью проверки равенства единице элементов одномерного массива, индексы которых кодируются с помощью описанного способа.

Участникам соревнования было предоставлено уже заданное предусловие верифицируемой программы. Предусловие программы задано на языке Applicative Common Lisp [68], языке задания спецификаций в системе C-lightVer. Отметим, что предусловие играет свою роль при задании инварианта цикла. Предусловие накладывает следующие ограничения на входные данные программы (листинг 5).

```
(and (integerp variable_count) (< 0 variable_count)
        (integer-listp implication_graph_transitive_closure)
        (= (len implication_graph_transitive_closure)
           (* 4 (* variable_count variable_count))))
```

Листинг 5. Предусловие функции “twosat_solver”.
Listing 5. Precondition of “twosat_solver” function.

Участникам соревнования было также предоставлено постусловие программы (листинг 6).

```
(implies (= satisfiable 0)
         (not (twosat-solver (boolean-variable-values
                                    variable_count nil)
                variable_count
                implication_graph_transitive_closure)))
```

Листинг 6. Постусловие функции “twosat_solver”.
Listing 6. Postcondition of “twosat_solver” function.

Данное постусловие описывает свойство, что если переменная satisfiable по итогу выполнения программы равна 0, то не выполнен предикат twosat-solver из теории предметной области задачи. Участникам соревнования предоставлялся также файл twosat-solver.lisp, содержащий теорию предметной области задачи, заданную на языке Applicative Common Lisp. В данном файле определен используемый в постусловии предикат twosat-solver. Данный предикат проверяет, является ли формула выполнимой, с помощью перебора всех возможных наборов значений переменных, аналогично проверке выполнимости формулы на таблице истинности. Сама формула передается предикату twosat-solver в виде транзитивного замыкания матрицы смежности графа импликаций. Также в теории предметной области содержится теорема twosat-unsat о том, что если для какой-либо переменной формулы выполнено важное для задачи 2-SAT условие, то применение предиката twosat-solver к этой формуле ложно, то есть формула невыполнима. Для решения задачи полезно то, что система доказательства ACL2, используемая в системе C-lightVer, может автоматически применять данную терему для автоматизации доказательства условий корректности.

Отметим, что постусловие описывает только случай, когда формула невыполнима, и не описывает случай, когда формула выполнима. То есть, участникам нужно было доказать выполнение “частичного” постусловия. Для этого участникам было достаточно задать “частичный” инвариант цикла, который позволяет доказать выполнение “частичного” постусловия. Еще одна особенность постусловия состоит в том, что данное постусловие позволяет в случае невыполнимости формулы в некотором смысле проверить эквивалентность двух реализаций: полиномиальную [62-63] (но нетривиальную) реализацию задачи 2-SAT с помощью графа импликаций и неполиномиальную (но тривиальную) реализацию поверки выполнимости на таблице истинности. Такую возможность

дедуктивной верификации по доказательству эквивалентности (в определенном случае) было полезно в образовательных целях показать участникам соревнования.

Организатором соревнования ожидалось решение в виде инварианта, представляющего собой конъюнкцию трех частей: ограничения на переменные (фактические, копия предусловия), нужное для доказательства условие выхода из цикла, ограничение на переменную-счетчик цикла, позволяющее доказать и условие входа в цикл, и условие продолжения итерации, и условие выхода из цикла, и ограничение на переменную-результат, важное для доказательства условия выхода из цикла. Листинг 7 демонстрирует самую нетривиальную часть инварианта цикла в виде ограничения на переменную-результат.

```
(implies (= satisfiable 0)
         (and(=(nth(+(*x(*variable_count 2)) (+ x variable_count))
                implication_graph_transitive_closure) 1)
             (=(nth(+(*(+x variable_count) (*variable_count 2)) x)
                implication_graph_transitive_closure) 1)
         (< x variable_count))))
```

Листинг 7. Часть инварианта цикла, являющаяся ограничением на переменную "satisfiable".
Listing 7. Loop invariant part that is constraint on "satisfiable" variable.

Данное ограничение описывает, что если переменная satisfiable равна нулю, то для i-той переменной формулы не выполняется важное для задачи 2-SAT условие. Данное ограничение позволяет доказать выполнение постусловия при завершении цикла с помощью применения в качестве леммы теоремы twosat-unsat из теории предметной области. Это позволяет провести автоматическую успешную верификацию решения в системе C-lightVer.

5.5 Задача 5. Построение и проверка модели протокола консенсуса IBFT

В задаче участникам предлагалось построить формальную модель протокола консенсуса IBFT и проверить её свойства в любом удобном средстве проверки модели. Организаторами было рекомендовано использование языка TLA+ и TLC Model Checker.

Протокол консенсуса – это алгоритм, решающий проблему достижения консенсуса в распределенных системах, когда все корректные процессы должны принять решение о некотором общем предлагаемом значении. Алгоритм IBFT (Istanbul Byzantine Fault Tolerance) [69] вдохновлен алгоритмом PBFT (Practical Byzantine Fault Tolerance) [70] и также решает проблему достижения консенсуса в распределенных системах. Ключевая особенность такого рода протоколов консенсуса заключается в их устойчивости к византийским ошибкам, предполагающим наличие вредоносных узлов, которые могут функционировать произвольно или умышленно вредить сети, передавая противоречивую или ложную информацию. Как и в случае с другими BFT-алгоритмами, необходимым условием достижения консенсуса в распределенной системе с N византийскими узлами, реализующей алгоритм IBFT, является общее число узлов, превышающее 3N [71].

Для алгоритма консенсуса обычно необходимо проверить 3 ключевых свойства:

- Согласованность: все корректно работающие узлы принимают одинаковое значение.
- Корректность: принятое значение было предложено одним из имеющихся узлов.
- Завершаемость: все корректно работающие узлы достигают определённого значения.

Выполнение этих свойств для протокола IBFT и предлагалось проверить участникам соревнования VeHa-2024. Реализовать модель алгоритма участники могли на основе псевдокода или описания алгоритма на естественном языке, предложенном в статье [69].

6. Обзор решений и ошибки, допущенные участниками

Приведем некоторые комментарии экспертов по ошибкам в решениях участников. Все решения можно найти в репозитории соревнования VeHa-2024 [23].

По задаче 1:

В решении [72] вместо описания behaviors (описание всех возможных сценариев поведения функции, требующее доказательства полноты) участники использовали описание ensures (простое описание постусловий), и вот почему: они выявили не все возможные случаи как в функции compute_mask, так и в функции check_permission – например, не доказывается постусловие (см. листинг 8) и ряд других случаев. (Всего было пропущено около 5 сценариев в базовой функции и 2 – в функции compute_mask).

```
ensures getSecurityLabel(current) !=\null && isMaxIlev(current) &&
                  currentIsFileSystemRoot && !isPublic(file) &&
                  (cmd == ORDINARY || cmd == RARE)
   ==>  \result == -NO_PERM
```

Листинг 8. Постусловие программы.
Listing 8. Postcondition of the program.

В решении [73] выявлены следующие ошибки:

1. Один из инвариантов цикла задан неполно – поэтому не доказывается условие Termination (условие окончания цикла);
2. Указаны не все возможные инварианты – в частности, нет инварианта с двумя переменными;
3. Код, исправляющий ошибку и заключающийся в проверке того, что метка безопасности – пустая, вставлен не в то место, где нужно (студенты добавили его после вызова функции, получающей данную метку, а надо было до). Поэтому совершенно справедливо не доказывается корректность вызова одной из подфункций, а именно slabel_ilev.

По задаче 2:

Из девяти зарегистрировавшихся участников на проверку поступили решения от пяти. Команды Aleksei_Volkov, Gleb, IvanSmirnov, orenty7 выполнили все задания. Команда 1 выполнила задания 1 и 4, но в задании 2 была допущена досадная ошибка – была определена не та структура решетки для целых чисел, эта оплошность не позволила закончить задание 3.

По задаче 3:

Решение [74]:

- Уровень 1. Хорошая организация решения, задача решается, код вычисления суммы похож на оригинальное OpenCL решение, с другой стороны, вставки нового кода немного чужеродные для авторского кода по названию переменных, пробелам и т.д. Модель работает, представленные результаты в отчете подтверждаются, из ошибок – не промоделирован барьер, решено синхронизацией между процессами через каналы.
- Уровень 2. Сделано сильно просто для заданной постановки задачи, не до конца разобрались с потоками исполнения (warp). Модель работает. Планировщик, распределяющий по потокам исполнения, не промоделирован.

Решение [75]:

- Уровень 1. Разобрались с кодом, предложили сделать исправление, но решение задачи так и не увиделось. Модель работает. Не удалось получить результаты, представленные в отчете. Нет моделирования задачи из задания (которая была на OpenCL).
- Уровень 2. Идеи с потоками исполнения достойные, но есть ошибки в реализации. Сделана попытка промоделировать планировщик потоков исполнения. Не доделано, исходная задача не промоделирована.
- Уровень 3. Очень много, конечно, разобрали по ветвлениям и маске, но невозможно согласиться с предложением плодить множество процессов в программах на языке Promela, дорогих для верификации.

По задаче 4:

Результаты решения данной задачи, как и решения всех остальных задач, доступны на сайте соревнования [23]. Из 5-ти зарегистрировавшихся индивидуальных участников и 6-ти зарегистрировавшихся команд решения сдали 3 индивидуальных участника и 2 команды. Все сдавшие решения отдельные участники и команды успешно освоили сложную предметную область дедуктивной верификации, а также задачи 2-SAT, и сдали решения, очень близкие к ожидаемому организатором решению. Поэтому все сдавшие решения участники и команды получили за свои решения полные 10 баллов. В комментариях в форме обратной связи по итогам соревнования участники отметили, что этому поспособствовало подготовленное организатором пособие, содержащее пример по заданию инварианта цикла. Кроме того, что участники и команды успешно изучили пособие, необходимо отметить, что один из участников ради обучения дедуктивной верификации еще до старта соревнования пытался верифицировать программы с циклами над массивами. Поэтому данный участник был отмечен почетной грамотой за волю к победе. Еще необходимо отметить принимавшего участие в соревновании известного специалиста по формальным методам в программировании. Данный специалист сдал вместе с решением полезное тестовое описание решения и своего мнения о задаче, за что был отмечен благодарственным письмом организаторов.

7. Результаты и обратная связь

По задаче 1 были объявлены 4 победителя (Дипломы 1 и 2 степени, благодарность, почетная грамота за освоение системы верификации Frama-C).

По задаче 2 были объявлены 5 победителей (4 диплома 1 степени и один диплом 2 степени), а также грамоты (за самое быстрое решение, за внимание к корректности функциональной спецификации, за самое короткое решение и использование автоматики, за выделение вспомогательных утверждений).

По задаче 3 были объявлены 2 победителя (две команды с дипломами за 1 и 2 место) + дополнительная почетная грамота “За найденную ошибку в исходном коде организаторов”.

По задаче 4 были объявлены 5 победителей: 4 победителя с дипломом 1 степени (команды и индивидуальные участники), одно благодарственное письмо за лучшее решение задач соревнования VeHa-2024 и, кроме этого, дополнительная почетная грамота за волю к победе.

По задаче 5 не было подано решений на проверку, соответственно, победители не были определены.

В итоге было объявлено 16 победителей (как индивидуальных участников, так и команд). Один из участников получил дипломы в трех номинациях. На рис. 4 представлен процесс вручения грамот и дипломов победителям и призерам.

Рис. 4. Очное вручение грамот и дипломов победителям и призерам соревнования VeHa-2024.
Fig. 4. In-person presentation of winner's certificates and certificates of honor to winners and prize-winners of VeHa-2024 contest.

По сравнению с прошлым соревнованием, резко выросло число регистраций, некоторые участники пробовали себя в решении разных задач, и даже меняли предварительные предпочтения, поскольку не могли решить заранее выбранную задачу. С другой стороны, решений было получено не так много, скорее всего, большинство участников просто прицеливалось и осваивало инструменты формальной верификации, поскольку обратной связи, почему они не отправили решение, получено не было. Еще одно важное отличие, что по сравнению с прошлым соревнованием, где в основном задания были достаточно абстрактные, в этот раз они стали более практическими, и решения таких задач с помощью формальных методов участвующие в соревнованиях были в состоянии сделать, а не просто усвоить предложенные инструменты. Также была выявлена проблема с использованием общего репозитория на GitHub для создания предложения исправления (pull-request) с решениями участников, заключающаяся в том, что некоторые участники отслеживали чужие решения и копировали их части. Возможно, в будущем необходимо создание выделенной системы отправки решений.

После проведения соревнования была запущена форма обратной связи, которую заполнили 6 человек. Результаты опроса показаны на рис. 5 – рис. 8.

8. Заключение

Мы считаем, что соревнование VeHa-2024 прошло успешно, с бóльшим вовлечением участников, чем на VeHa-2023. Решение предложить участникам несколько разнородных задач на выбор и привлечение компаний к формированию заданий подняло состязание на следующий уровень.

Индустриальные задачи позволили продемонстрировать применимость формальных методов, разобраться с классом их возможных применений, а также показать участникам

потенциальные перспективные задачи для будущей работы в той области, о которой, возможно, они не предполагали ранее. По результатам состязания победителям была предложена стажировка в одной из компаний, организаторов соревнования.

При подготовке такого рода соревнований образовательная составляющая играет немаловажное значение. В вузах формальные методы совмещаются либо с курсами по тестированию программного обеспечения [76], либо преподаются как спецкурсы или курсы по выбору, что недостаточно большой аудитории программистского сообщества. При проведении соревнований возможно улучшить компетенции разработчиков, тех, кому это действительно необходимо, в форме лекций и обучающих материалов, а также продемонстрировать свои разработки в виде средств формальной верификации заинтересованной аудитории.

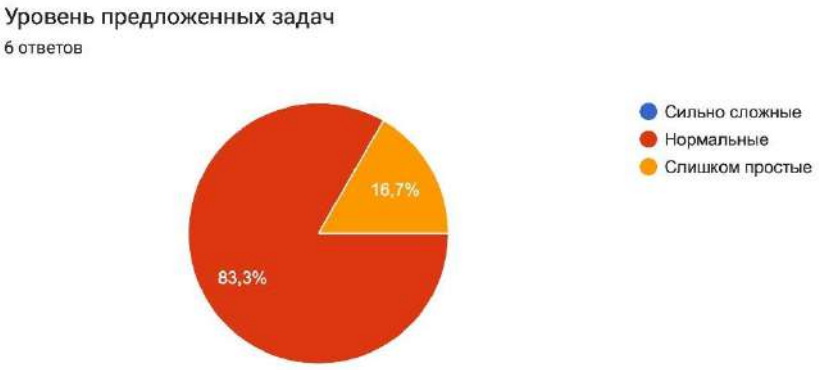


Рис. 5. Распределение отзывов по уровню предложенных задач.
Fig. 5. Distribution of survey responses by proposed tasks levels.

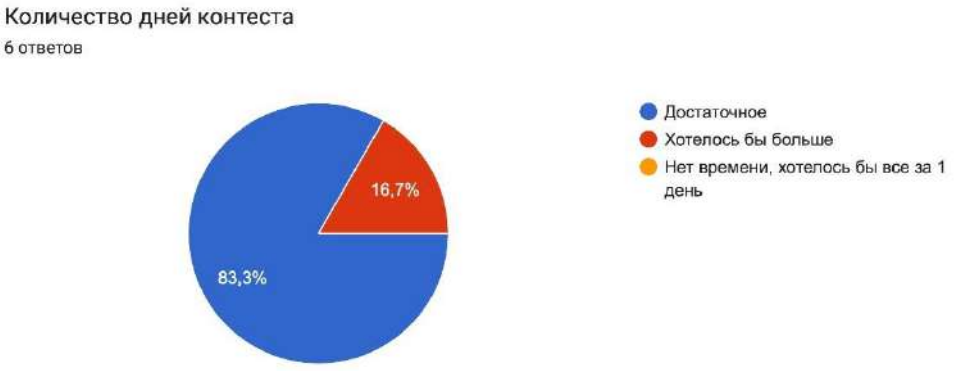


Рис. 6. Распределение отзывов по количеству дней соревнования.
Fig. 6. Distribution of survey responses by quantity of contest days.

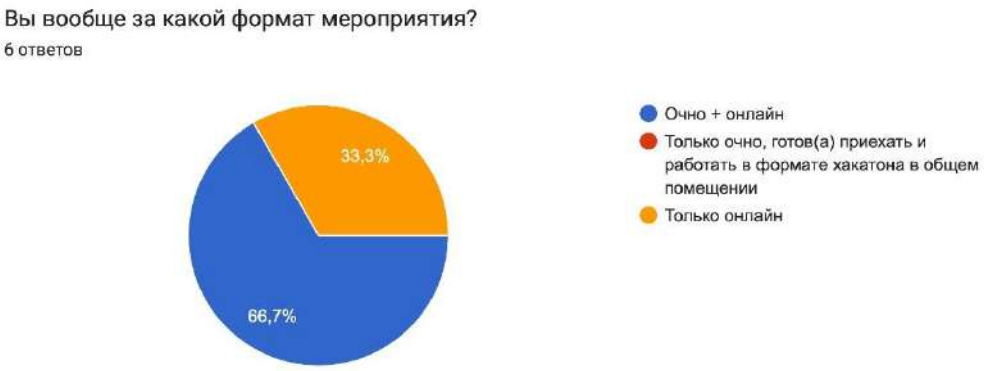


Рис. 7. Распределение отзывов на формат мероприятия.
Fig. 7. Distribution of survey responses by event format.

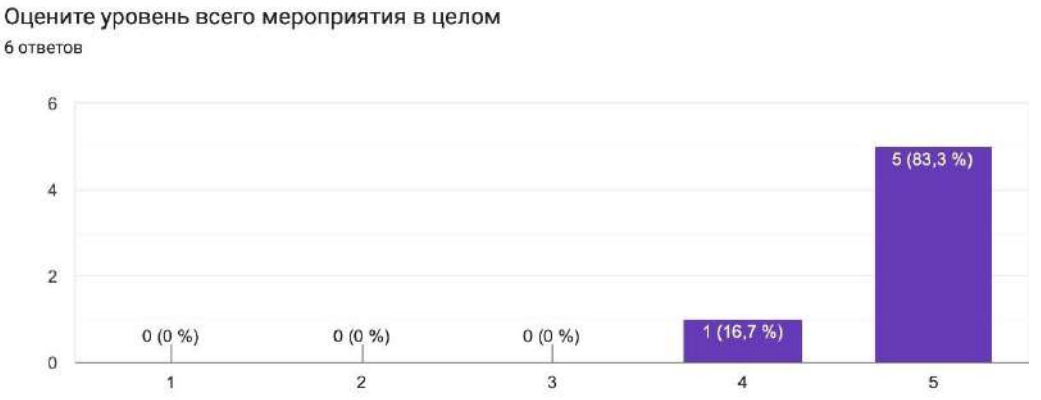


Рис. 8. Распределение оценок мероприятия.
Fig. 8. Distribution of survey responses by event ratings.

Список литературы / References

[1]. Старолетов С.М., Кондратьев Д.А., Гаранина Н.О., Шошмина И.В. Соревнования по формальной верификации VeHa-2023: опыт проведения // Труды Института системного программирования РАН. 2024, т. 36, № 2, с. 141-168.

[2]. Bartocci E., Beyer D, Black P. E., Fedyukovich G., Garavel H., Hartmanns A., Huisman M., Kordon F., Nagele J., Sighireanu M., Steffen B., Suda M., Sutcliffe G., Weber T., Yamada A. TOOLympics 2019: An overview of competitions in formal methods. In Proc. Tools and Algorithms for the Construction and Analysis of Systems, ser. LNCS. Springer, 2019, vol. 11429, pp. 3–24.

[3]. Beyer D., Huisman M., Kordon F., Steffen B. TOOLympics II: competitions on formal methods. International Journal on Software Tools for Technology Transfer, vol. 23, no. 6, pp. 879–881, 2021.

[4]. TOOLympics Challenge 2023 / Ed. by D. Beyer, A. Hartmanns, F. Kordon. – Cham: Springer, 2025. – 172 p.

[5]. Dross C., Furia C. A., Huisman M., Monahan R., Müller P. VerifyThis 2019: a program verification competition. International Journal on Software Tools for Technology Transfer, vol. 23, no. 6, pp. 883–893, 2021.

[6]. Ernst G., Huisman M., Mostowski W., Ulbrich M. VerifyThis – verification competition with a human factor. In Proc. Tools and Algorithms for the Construction and Analysis of Systems. Ser. LNCS. Springer, 2019, vol. 11429, pp. 176-195.
[7]. Denis X., Siegel S.F. VerifyThis 2023: An International Program Verification Competition. In TOOLympics Challenge 2023. TOOLympics 2024. Ser. LNCS. Springer, 2025, vol. 14550, pp. 147-159.
[8]. Jasper M., Mues M., Murtovi A., Schlüter M., Howar F., Steffen B., Schordan M., Hendriks D., Schiffelers R., Kuppens H., Vaandrager F. W. RERS 2019: Combining synthesis with real-world models. In Tools and Algorithms for the Construction and Analysis of Systems, ser. LNCS. Springer, 2019, vol. 11429, pp. 101–115.
[9]. Howar F., Jasper M., Mues M., Schmidt D., Steffen B. The Rers challenge: towards controllable and scalable benchmark synthesis. International Journal on Software Tools for Technology Transfer, vol. 23, no. 6, pp. 917– 930, 2021.
[10]. Giesl J., Rubio A., Sternagel C., Waldmann J., Yamada A. The termination and complexity competition. In Proc. Tools and Algorithms for the Construction and Analysis of Systems, ser. LNCS. Springer, 2019, vol. 11429, pp. 156–166.
[11]. Ahrendt W., Herber P., Huisman M., Ulbrich M. SpecifyThis – bridging gaps between program specification paradigms. In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles, ser. LNCS. Springer, 2022, vol. 13701, pp. 3-6.
[12]. Ernst G., Herber P., Huisman M., Ulbrich M. SpecifyThis Bridging Gaps Between Program Specification Paradigms: Track Introduction. In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification. Ser. LNCS. Springer, 2025, vol. 15221, pp. 3-7.
[13]. Jacobs B., Kiniry J. Warnier M. Java program verification challenges. In Proc. Formal Methods for Components and Objects, ser. LNCS. Springer, 2003, vol. 2852, pp. 202–219.
[14]. Leavens G. T., Leino K. R. M., Müller P. Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing, vol. 19, no. 2, pp. 159– 189, 2007.
[15]. Leino K. R. M., Moskal M. VACID-0: verification of ample correctness of invariants of data-structures, edition 0. In Proc. of Tools and Experiments Workshop at VSTTE, 2010.
[16]. Weide B. W., Sitaraman M., Harton H. K., Adcock B., Bucci P., Bronish D., Heym W. D., Kirschenbaum J., Frazier D. Incremental benchmarks for software verification tools and techniques. In Proc. Verified Software: Theories, Tools, Experiments, ser. LNCS. Springer, 2008, vol. 5295, pp. 84–98.
[17]. Geske M., Jasper M., Steffen B., Howar F., Schordan M., van de Pol J. Rers 2016: Parallel and sequential benchmarks with focus on LTL verification. In Proc. Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications, ser. LNCS. Springer, 2016, vol. 9953, pp. 787–803.
[18]. Beyer D., Huisman M., Klebanov V., Monahan R. Evaluating software verification systems: Benchmarks and competitions (Dagstuhl reports 14171). Dagstuhl Reports, vol. 4, no. 4, pp. 1–19, 2014.
[19]. VerifyThis Archive, 2024. Available at: https://www.pm.inf.ethz.ch/research/verifythis/Archive.html, accessed Nov 06, 2024.
[20]. VeHa contest, 2023. Available at: https://sites.google.com/view/veha23/, accessed Jul 06, 2024.
[21]. VeHa2023 on GitHub. Available at: https://github.com/VeHaContest/VeHa2023, accessed Jul 06, 2024.
[22]. VeHa contest, 2024. Available at: https://sites.google.com/view/veha2024, accessed Nov 06, 2024.
[23]. VeHa2024 on GitHub. Available at: https://github.com/VeHaContest/VeHa2024, accessed Jul 06, 2024
[24]. NSU profile "Formal methods of program and system analysis". Available at https://education.nsu.ru/formal-analysis-methods/, accessed Dec 20, 2024.
[25]. Khazeev M., Aslam H., de Carvalho D., Mazzara M., Bruel JM., Brown J.A. Reflections on Teaching Formal Methods for Software Development in Higher Education. In: Proc. Frontiers in Software Engineering Education, ser. LNCS. Springer, 2020, vol 12271, pp. 28–41.
[26]. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C program verification based on mixed axiomatic semantics. Automatic Control and Computer Sciences, vol. 48, no. 7, pp. 407 – 414, 2014.
[27]. Kondratyev D. A., Promsky A. V. Developing a self-applicable verification system. Theory and practice. Automatic Control and Computer Sciences, vol. 49, no. 7, pp. 445–452, 2015.
[28]. Kondratyev D. A., Maryasov I. V., Nepomniaschy V. A. The automation of C program verification by the symbolic method of loop invariant elimination. Automatic Control and Computer Sciences, vol. 53, no. 7, pp. 653–662, 2019.
[29]. Kondratyev D. A., Nepomniaschy V. A. Automation of C program deductive verification without using loop invariants. Programming and Computer Software, vol. 48, no. 5, pp. 331–346, 2022.

[30]. Khazeev M., Mazzara M., Aslam H., de Carvalho D. Towards a Broader Acceptance of Formal Verification Tools. In: Proc. Impact of the 4th Industrial Revolution on Engineering Education, ser. Advances in Intelligent Systems and Computing. Springer, 2020, vol. 1135, pp. 188–200.

[31]. Klebanov V., Müller P., Shankar N., Leavens G. T., Wüstholz V., Alkassar E., Arthan R., Bronish D., Chapman R., Cohen E., Hillebrand M., Jacobs B., Leino K. R. M., Monahan R., Piessens F., Polikarpova N., Ridge T., Smans J., Tobies S., Tuerk T., Ulbrich M., Weiß B. The 1st verified software competition: Experience report. In Proc. FM 2011: Formal Methods, ser. LNCS. Springer, 2011, vol. 6664, pp. 154 – 168.

[32]. Filliâtre J.-C., Paskevich A., Stump A. The 2nd verified software competition: Experience report. In Proc. of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, ser. CEUR Workshop Proceedings. RWTH Aachen University, 2012, vol. 873, pp. 36–49.

[33]. Bormer T., Brockschmidt M., Distefano D., Ernst G., Filliâtre J.-C., Grigore R., Huisman M., Klebanov V., Marché C., Monahan R., Mostowski W., Polikarpova N., Scheben C., Schellhorn G., Tofan B., Tschannen J., Ulbrich M. The COST IC0701 verification competition 2011. In Proc. Formal Verification of Object-Oriented Software, ser. LNCS. Springer, 2012, vol. 7421, pp. 3–21.

[34]. Huisman M., Klebanov V., Monahan R. VerifyThis 2012. International Journal on Software Tools for Technology Transfer, vol. 17, no. 6, pp. 647–657, 2015.

[35]. Huisman M., Klebanov V., Monahan R., Tautschnig M. VerifyThis 2015. International Journal on Software Tools for Technology Transfer, vol. 19, no. 6, pp. 763–771, 2017.

[36]. Huisman M., Monti R., Ulbrich M., Weigl A. The VerifyThis collaborative long term challenge. In Proc. Deductive Software Verification: Future Perspectives, ser. LNCS. Springer, 2020, vol. 12345, pp. 246 – 260.

[37]. Ernst G. Weigl A. Verify This: Memcached–a practical long-term challenge for the integration of formal methods. In Proc. iFM 2023, ser. LNCS. Springer, 2024, vol. 14300, pp. 82–89.

[38]. Ahrendt W., Ernst G., Herber P., Huisman M., Monti R.E., Ulbrich M., Weigl A. The VerifyThis Collaborative Long-Term Challenge Series. In TOOLympics Challenge 2023. TOOLympics 2024. Ser. LNCS. Springer, 2025, vol. 14550, pp. 160-170.

[39]. Hoare C. A. R., Leavens G. T., Shankar N. The verified software initiative: A manifesto. ACM Computing Surveys, vol. 41, no. 4, pp. 1–8, 2009.

[40]. Müller P., Shankar N. The first fifteen years of the verified software project. In Proc. Theories of Programming: The Life and Works of Tony Hoare, 2021, pp. 93–124.

[41]. Huisman M., Klebanov V., Monahan R. On the organisation of program verification competitions. In Proc. of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, ser. CEUR Workshop Proceedings. RWTH Aachen University, 2012, vol. 873, pp. 50–59.

[42]. Paul M., Haslbeck L., Wimmer S. Competitive Proving for Fun. In Selected Student Contributions and Workshop Papers of LuxLogAI 2018, vol. 10, pp. 9-14.

[43]. Paulin-Mohring C. Introduction to the Coq Proof-Assistant for Practical Software Verification. In Proc. Tools for Practical Software Verification. Ser. LNCS. Springer, 2012, vol. 7682, pp. 45-95.

[44]. Jasper M., Fecke M., Steffen B., Schordan M., Meijer J., van de Pol J., Howar F., Siegel S. F. The Rers 2017 challenge and workshop (invited paper). In Proc. of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, 2017, pp. 11–20.

[45]. Jasper M., Mues M., Schlüter M., Steffen B, Howar F. Rers 2018: CTL, LTL, and reachability. In Proc. Leveraging Applications of Formal Methods, Verification and Validation of Systems, ser. LNCS. Springer, 2018, vol. 11245, pp. 433-447.

[46]. Marché C., Zantema H. The termination competition. In Proc. Term Rewriting and Applications, ser. LNCS. Springer, 2007, vol. 4533, pp. 303–313.

[47]. Giesl J, Mesnard F., Rubio A., Thiemann R., Waldmann J. Termination competition (TermCOMP 2015). In Proc. Automated Deduction - CADE-25, ser. LNCS. Springer, 2015, vol. 9195, pp. 105–108.

[48]. Yamada A. Termination of term rewriting: Foundation, formalization, implementation, and competition. In Proc. 8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023), ser. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum fur Informatik, 2023, vol. 260, pp. 4:1–4:5.

[49]. The International Collegiate Programming Contest, ICPC. Available at https://icpc.global/, accessed Dec 21, 2024.

[50]. Chas K. The World's Smartest Programmers Compete: ACM ICPC. Communications of the ACM. Jul 2 2013. Available at https://cacm.acm.org/blogs/blog-cacm/165692-the-worlds-smartest-programmers-compete-acm-icpc/, accessed Dec 21, 2024.

[51]. Slamecka V.(editor) Proceedings of the 5th annual ACM computer science conference, CSC 1977, Atlanta, Georgia, USA, January 31 – February 2, 1977. ACM 1977,

https://doi.org/10.1145/800008.808038.
[52]. Workshop "Program Semantics, Specification and Verification: Theory and Applications". Available at https://persons.iis.nsk.su/ru/PSSV-2024, accessed Dec 20, 2024.
[53]. Baudin P., Filliâtre J. C., Marché C., Monate B., Moy Y., & Prevosto V. (2008). Acsl: Ansi c specification language. CEA-LIST, Saclay, France, Tech. Rep. v1, 2.
[54]. Smalley S., Vance C., & Salamon W. (2001). Implementing SELinux as a Linux security module. NAI Labs Report, 1(43), 139.
[55]. Garanina N., Staroletov S., Gorlatch S. (2022, September). Model Checking Meets Auto-Tuning of High-Performance Programs. In International Symposium on Logic-Based Program Synthesis and Transformation (pp. 63-82). Cham: Springer International Publishing.
[56]. Garanina N., Staroletov S., Gorlatch S. Auto-Tuning High-Performance Programs Using Model Checking in Promela //arXiv preprint arXiv:2305.09130. – 2023.
[57]. GPGU Sim, manual. Revision 1.2 (GPGPU-Sim 3.1.1). Ed.: Tor M. Aamodt, Wilson W.L. Fung, Tayler H.Hetherington, http://gpgpu-sim.org/manual/index.php/Main_Page# Fetch_and_Decode_Software_Model, accessed Nov 06, 2024.
[58]. Lindholm E., Nickolls J., Oberman S., & Montrym J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro, 28(2), 39-55.
[59]. Whitepaper “NVIDIA Tesla P100”. https://www.techpowerup.com/gpu-specs/docs/nvidia-gp100-architecture.pdf, accessed Nov 06, 2024.
[60]. Furia C. A., Meyer B., Velder S. Loop invariants: Analysis, classification, and examples. ACM Computing Surveys, vol. 46, no. 3, pp. 1–51, 2014.
[61]. Ernst G. Loop verification with invariants and contracts. In Proc. Verification, Model Checking, and Abstract Interpretation, ser. LNCS. Springer, 2022, vol. 13182, pp. 69–92.
[62]. Krom M.R. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, vol. 13, no. 1-2, pp. 15-20, 1967.
[63]. Aspvall B., Plass M.F., Tarjan R.E. A linear-time algorithm for testing the truth of certain quantified boolean formulas. Information Processing Letters, vol. 8, no. 3, pp. 121-123, 1979.
[64]. Myreen M.O., Davis J. The Reflective Milawa Theorem Prover Is Sound. In Proc. Interactive Theorem Proving. Ser. LNCS. Springer, 2014, vol. 8558, pp. 421-436.
[65]. Blanchette J.C., Fleury M., Lammich P, Weidenbach C. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. Journal of Automated Reasoning, vol. 61, no. 1, pp. 333–365, 2018.
[66]. Fleury M., Blanchette J.C., Lammich P. A verified SAT solver with watched literals using imperative HOL. In Proc. of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018, pp. 158–171.
[67]. Ciobâcă Ș., Andrici C.-C. A Verified Implementation of the DPLL Algorithm in Dafny. Mathematics, vol. 10, no. 13, article id: 2264, 2022.
[68]. Moore J. S. Milestones from the pure Lisp theorem prover to ACL2. Formal Aspects of Computing, vol. 31, no. 6, pp. 699–732, 2019.
[69]. Moniz H. (2020). The Istanbul BFT consensus algorithm. arXiv preprint arXiv:2002.03613.
[70]. Castro M., Liskov B. Practical Byzantine Fault Tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999.
[71]. Pease M., Shostak R., Lamport L. Reaching Agreement in the Presence of Faults. Association for Computing Machinery, vol. 27, no. 2, 1980.
[72]. https://github.com/VeHaContest/VeHa2024/tree/main/solution_1_re_tofl, accessed Nov 06, 2024.
[73]. https://github.com/VeHaContest/VeHa2024/tree/main/solution_1_DevTools_Itmo, accessed Nov 06, 2024.
[74]. https://github.com/VeHaContest/VeHa2024/tree/main/solution_3_RARe, accessed Nov 06, 2024.
[75]. https://github.com/VeHaContest/VeHa2024/tree/main/solution_3_power_o%60_nine, accessed Nov 06, 2024.
[76]. Staroletov S. Teaching the discipline “Software Testing and Verification” to future programmers. System Informatics. vol. 21, pp. 1-28, 2022.

Информация об авторах / Information about authors

Дмитрий Александрович КОНДРАТЬЕВ – кандидат физико-математических наук, научный сотрудник ИСИ СО РАН, старший преподаватель НГУ. Сфера научных интересов: формальная верификация, дедуктивная верификация, логика Хоара и автоматическое доказательство теорем.

Dmitry Alexandrovich KONDRATYEV – Cand. Sci. (Phys.-Math.), researcher at Ershov Institute of Informatics Systems Siberian Branch of the RAS, senior lecturer at Novosibirsk State University. Research interests: formal verification, deductive verification, Hoare logic and automated theorem proving.

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент (ВАК). Сфера научных интересов: формальная верификация, проверка моделей, киберфизические системы, операционные системы.

Sergey Mikhailovich STAROLETOV – Cand. Sci. (Phys.-Math.), associate professor. Research interests: formal verification, model checking, cyber-physical systems, operating systems.

Ирина Владимировна ШОШМИНА – кандидат технических наук, доцент. Сфера научных интересов: формальная верификация, распределенные системы и алгоритмы.

Irina Vladimirovna SHOSHMINA – Cand. Sci. (Tech.), associate professor. Research interests: formal verification, model checking, distributed systems and algorithms.

Анастасия Владимировна КРАСНЕНКОВА – специалист по анализу безопасности РусБИТех-Астра. Сфера научных интересов: формальная верификация, дедуктивная верификация, программирование, анализ безопасности, математическая логика, системы распределённого реестра.

Anastasiya Vladimirovna KRASNENKOVA – security analyst at RusBITech-Astra. Research interests: formal verification, deductive verification, programming, security analysis, mathematical logic, distributed ledger systems.

Кирилл Викторович ЗИБОРОВ – инженер Positive Technologies, аспирант МГУ им. М. В. Ломоносова. Сфера научных интересов: формальная верификация, проверка моделей, распределенные системы и алгоритмы, системы распределённого реестра.

Kirill Viktorovich ZIBOROV – formal verification engineer at Positive Technologies, postgraduate student at Lomonosov Moscow State University. Research interests: formal verification, model checking, distributed systems and algorithms, distributed ledger systems.

Николай Вячеславович ШИЛОВ – кандидат физико-математических наук, доцент (ВАК), руководитель Лаборатории программной инженерии Университета Иннополис. Сфера научных интересов: прикладная логика, основания математики и программирования, преподавание фундаментальной математики и теории программирования

Nikolay Vyacheslavovich SHILOV – Cand. Sci. (Phys.-Math.), associate professor, head of the Laboratory of Software and Service Engineering of Innopolis University. Research interests: applied logic, foundations of Mathematics and Programming, Mathematical and Theory of Programming education and teaching

Наталья Олеговна ГАРАНИНА – кандидат физико-математических наук, старший научный сотрудник ИСИ СО РАН, старший научный сотрудник ИАиЭ СО РАН, ведущий научный сотрудник ООО ОКП «АРС», доцент НГУ. Сфера научных интересов: формальная верификация, проверка моделей, системы реального времени, инженерия требований.

Natalia Olegovna GARANINA – Cand. Sci. (Phys.-Math.), senior researcher at Ershov Institute of Informatics Systems Siberian Branch of the RAS, associate professor at Novosibirsk State University. Research interests: formal verification, model checking, distributed systems, requirements engineering.

Тимофей Юрьевич ЧЕРГАНОВ – старший специалист по анализу безопасности РусБИТех-Астра. Сфера научных интересов: формальная верификация, компьютерная безопасность.

Timofey Yuryevich CHERGANOV – senior security analyst at RusBITech-Astra. Research interests: formal verification, computer security.

DOI: 10.15514/ISPRAS-2025-37(1)-11

CC BY

Использование технологий искусственного интеллекта для проведения психологического тестирования

Д.Д. Григорьева, ORCID: 0009-0006-9211-7012 <g.darya.work@gmail.com>
Д.В. Серов, ORCID: 0009-0001-2362-1075 <serovd33@gmail.com>
Д.С. Сорокин, ORCID: 0009-0000-2166-5974 <dsorokin.job@yandex.ru>
А.И. Мартышкин, ORCID: 0000-0002-3358-4394 <mai@penzgtu.ru>

Пензенский государственный технологический университет,
440039, Россия, г. Пенза, проезд Байдукова/ул. Гагарина, д. 1а/11.

Аннотация. Данная исследовательская работа посвящена использованию технологий искусственного интеллекта для проведения теста Роршаха. Рассматриваются методы машинного обучения. Оба этих метода используется для многоклассовой классификации категорий ответов. В статье описаны алгоритмы методов машинного обучения для интерпретации результатов, алгоритм выставления результатов по одной из категорий и конечного результата тестирования в веб-интерфейсе для пользователя. Применение искусственного интеллекта для проведения проективных методик тестирования, на примере теста Роршаха, открывает новые возможности для самодиагностики и терапии.

Ключевые слова: нейронная сеть; многоклассовая классификация; машинное обучение; психологическое тестирование; тест Роршаха; наивный Байесовский классификатор; язык Python; библиотека TensorFlow; библиотека scikit-learn; библиотека Keras; большие языковые модели LLM; инфраструктура langchain.

Для цитирования: Григорьева Д.Д., Серов Д.В., Сорокин Д.С., Мартышкин А.И. Использование технологий искусственного интеллекта для проведения психологического тестирования. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 185–200. DOI: 10.15514/ISPRAS–2025–37(1)–11.

Using Artificial Intelligence Technologies to Conduct Psychological Testing

D.D. Grigoreva, ORCID: 0009-0006-9211-7012 <g.darya.work@gmail.com>
D.V. Serov, ORCID: 0009-0001-2362-1075 <serovd33@gmail.com>
D.S. Sorokin, ORCID: 0009-0000-2166-5974 <dsorokin.job@yandex.ru>
A.I. Martyshkin, ORCID: 0000-0002-3358-4394 <mai@penzgtu.ru>

Penza State Technological University,
1a/11, Baidukova Passage/Gagarin st., Penza, 440039, Russia.

Abstract. This research paper focuses on the use of artificial intelligence technologies for the Rorschach test. Machine learning techniques are considered. Both of these methods are used for multi-class classification of response categories. The paper describes the algorithms of machine learning and deep learning methods for interpreting the results, the algorithm for scoring one of the categories and the final test result in the web interface for the user. The application of artificial intelligence for projective testing techniques, on the example of the Rorschach test, opens new opportunities for self-diagnosis and therapy.

Keywords: neural network; multiclass classification; machine learning; psychological testing; Rorschach test; naive Bayes classifier; Python; TensorFlow; scikit-learn; Keras; LLM; langchain.

For citation: Grigoreva D.D., Serov D.V., Sorokin D.S., Martyshkin A.I. Using artificial intelligence technologies to conduct psychological testing. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 185-200 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-11.

1. Введение

Тест Роршаха, также известный как пятна Роршаха, представляет собой проективный психологический тест из 10 карточек, представленных на рис. 1, используемый для оценки личности и эмоционального состояния человека. Тест используется в различных областях психологии и психиатрии:

- В клинической психологии и психиатрии для выявления признаков таких расстройств как шизофрения, депрессия, тревожное расстройство личности. Также для оценки эмоционального состояния пациента при ведении или планировании терапии;
- В судебной психологии для проведения экспертизы психического состояния обвиняемых и свидетелей, определения вменяемости;
- Для оценки эмоционального и психического состояния детей и подростков, поскольку ребёнок не всегда точно может описать проблему, градуировать свои эмоции и причину их возникновения;
- В научных исследованиях для изучения особенностей личности и эмоциональных реакций;
- Для оценки личностных качеств сотрудников при приёме на работу или для оценки их готовности к выполнению определённых задач, поскольку тест Роршаха практически невозможно обмануть;
- Для оценки психологической устойчивости и эмоционального состояния военнослужащих [1].

Тест Роршаха является одним из наиболее эффективных и часто используемых проективных тестов [2].

Рис. 1. Карточки теста Роршаха.
Fig. 1. Rorschach test cards.

2. Процесс проведения теста Роршаха

Методика теста основана на интерпретации испытуемым серии абстрактных чернильных пятен, что позволяет выявить скрытые аспекты психики. Тест проводится в несколько этапов:

1) Исследователь готовит 10 карточек с симметричными черно-белыми и цветными чернильными пятнами.
2) Тестируемому объясняют, что нет списка верных ответов, и что он должен описывать то, что видит на карточках. Исследователь также не имеет права подсказывать или намекать на ответ. Также при проведении тестирования не должно быть никаких отвлекающих факторов.
3) Карточки показываются поочередно, обычно в определенной последовательности.
4) Тестируемого просят описывать, что он видит в пятнах, какие ассоциации возникают. Исследователь записывает ответы пациента, включая время реакции и любые дополнительные комментарии, такие как отказ от ответа, дополнительный ответ, шоковая реакция.
5) После завершения первого этапа исследователь может задать дополнительные вопросы для уточнения восприятий пациента [3].

После проведения тестирования, ответы шифруются и анализируются по следующим параметрам:

- Локация. Какая часть пятна была использована для ответа?
- Чёткость ответа. Изображение чётко распознаётся или границы размыты и приходится додумывать?
- Содержание. Какие объекты или сцены видит тестируемый?
- Оригинальность и популярность ответов. Часто ли встречаются такие ответы у других людей?
- Кинестетические показатели. Находится ли увиденный объект в движении?

Ответы интерпретируются с учетом теорий личности и психопатологии. Исследователь использует специальные таблицы и схемы для оценки различных аспектов личности пациента (например, когнитивные процессы, эмоциональные реакции, межличностные отношения) для более точной оценки личности или постановки диагноза [4].

Результаты теста оформляются в виде отчета, включающего анализ всех полученных данных и выводы о личности и эмоциональном состоянии пациента.

Отчет может включать рекомендации по дальнейшей терапии или психологическому сопровождению. Тест Роршаха является сложным диагностическим инструментом,

требующим высокого уровня подготовки и опыта от психолога. Интерпретация результатов с постановкой диагноза должна проводиться квалифицированным специалистом [5].

2.1 Определение требуемого функционала

Аналоги для проведения самодиагностики или оценки личности без участия специалиста, имеющиеся в свободном доступе дают возможность выбрать одну из категорий для выбора ассоциаций. Данные тесты можно рассматривать исключительно как развлекательные, так как наличие выбора вариантов ответа противоречит логике проективных методик тестирования. Наша задача - создать приложение, которое может быть использовано в рамках профессионального тестирования. Одним из важных критериев программы является наличие возможности ввода свободного ответа на карточку. Для интерпретации ответа, последующей шифрации и подсчёта данных используются нейронные сети.

Наша программа не может ставить диагнозы, поскольку для этого необходима консультация специалиста или заключение медицинской экспертизы, однако она может дать оценку личности по следующим категориям:

- Общая оценка личности. Высокие показатели по некоторым детерминантам имеют свой психологический смысл.
- Тип переживания. Определяет глобальных характер взаимодействия человека с внешним миром. В зависимости от типа переживания, человек использует разные паттерны восприятия произведений искусства. Тип переживания также характеризует богатство воображения, интенсивность рефлексии при восприятии визуального материала, склонность к эмпатии.
- Механизмы защиты. Конфликт, который диагностируется с помощью теста Роршаха, может иметь различную природу: эндо- или экзогенную, примерами конфликта могут быть апатия, стресс, тревожность или депрессия. В самом общем виде внешний конфликт порождается противоречием между собственными потребностями и социальной необходимостью. Также внутри системы может вызываться конфликт с помощью столкновений противоположных тенденций. В обоих случаях средствами разрешения конфликта будут выступать механизмы контроля и защиты, которые также прогнозирует наша программа.
- Интеллектуальные возможности. Тест Роршаха не даёт оценку интеллектуальным способностям, однако могут дать оценку гибкости ума и познавательным способностям тестируемого [1].

3. Описание решения задачи

Приложение для проведения теста Роршаха с помощью нейронных сетей, включает использование модели для категоризации популярных ответов, модели категоризации ответа по содержанию, методы для подсчета, интерпретации результата и отображения его на стороне пользователя (веб-интерфейс).

3.1 Модель категоризации популярных ответов

Ответы на тест Роршаха имеют две крайности: популярность и оригинальность. Под популярными Роршах подразумевал те ответы, которые даются каждым третьим испытуемым. Список популярных ответов фиксированный, но разный для каждой карточки. Список представлен в табл. 1.

Для поиска ответа из категории популярных не получится воспользоваться методом поиска `search()`, поскольку он не учитывает различные формы слова, такие как склонение, а также

синонимы и однокоренные слова другой части речи [6]. Эту проблему решает лемматизация, однако она не учитывает синонимы, например «собака» и «пёс».

Табл. 1. Популярные ответы на карточки.
Table 1. Popular card answers.

Номер карточки	Популярные ответы
1	Летучая мышь, бабочка, птица, человек
2	Медведь, слон, собака, человек
3	Человек
4	Летучая мышь, бабочка
5	Летучая мышь, бабочка, птица
6	Черепаха
7	Облако
8	Собака, ящерица, тигр
9	Голова человека
10	Краб, осьминог, паук, собака, кролик, заяц, лев

Использование наивного Байесовского классификатора будет наиболее рациональным, так как этот алгоритм машинного обучения используется для многоклассовой классификации и не допускает ошибок, которые могут быть при использовании `search()` или лемматизации. Он работает с высокой скоростью и прост в реализации [7].

Наивный Байес – модель, основанная на описанной выше теореме Байеса, имеет вид, показанный в формуле (1) [8].

$$c^* = arg_{c_j \in C} max \left[log\, P(c_j) + \sum_{i=1}^{X} P(x_i|c_j) \right] \quad (1)$$

где x – объект (слово), принадлежащий множеству X – словарь (в случае классификации текста), который описывается признаками 1...i, а $P(x_i|c_j)$ – плотность распределения этого признака для класса C.

Появление нулевых вероятностей можно предотвратить, добавив в формулу лапласовское сглаживание – alpha. Суть его добавления в формулу заключается в том, что мы притворяемся, что видели данное слово или признак на alpha раз больше, что повышает правильность предсказания [9].

Для классификации текста по категориям популярных ответов используем наивный Байесовский классификатор, который определяет введённый пользователем ответ на принадлежность к одной из следующих категорий:

- человек,
- голова человека,
- летучая мышь,
- медведь,
- бабочка,
- ящерица,
- птица,

- черепаха,
- слон,
- кролик,
- собака,
- осьминог,
- облако,
- тигр,
- паук,
- краб,
- заяц,
- лев [10].

Ответ пользователя предобратывается (выполняется удаление стоп-слов и пунктуации, лемматизации и токенизация) и передается в модель. Если правильность предсказания выше порогового уровня, то ответу присваивается категорий. Полученная категория, сравнивается со списком популярных ответов для карточки, так как для каждой карточки список популярных ответов индивидуален. Счетчик популярных ответов увеличивается, если результат положительный [11].

Для классификации был использован комплиментарный наивный Байесовский классификатор (ComplementNB) – это улучшенный вариант классического многоклассового классификатора MultinomialNB. Инициализация классификатора представлена в листинге 1. Этот вариант хорошо подходит для несбалансированного набора данных, а также быстро работает и обучается. ComplementNB оценивает нормированный вес признака для данного класса вместо вероятности наличия признака при заданном классе, как это делает MultinomialNB [12].

```
# Инициализация векторизатора
vectorizer = TfIdfVectorizer()
# Преобразование текста в числовые векторы
x_train = vectorizer.fit_transform(texts)
# Инициализация наивного байеса
clf = ComplementNB(alpha=0.1)
# Обучение
clf.fit(x_train, labels)
```

Листинг 1. Инициализация наивного Байесовского классификатора.
Listing 1. Initialization of a naive Bayesian classifier.

После определения архитектуры и выбора подходящего типа классификатора, необходимо перейти к данным. Набор данных для обучения и валидации состоит из набора предложений по каждой из категорий популярного ответа. Наш набор данных небольшой, поэтому для чтения его в массив Dataframe мы используем библиотеку для анализа и обработки данных pandas. Также, во избежание ошибок, следует указать кодировку при загрузке, по умолчанию – UTF-8. После загрузки следует разделить данные на обучающий и валидационный наборы. Для оценки производительности модели использовали метрику правильности (accuracy) [13].

На рис. 2 видно, что процент правильных ответов по категориям, при первом запуске классификатора до обучения, составил примерно 0.7.

В нашем случае обучение модели заключается в подборе гиперпараметра alpha, рассмотрим правильность работы модели в пределах alpha от 0.1 до 10. Как видно из графика на рис. 3, модель показывает наибольшую правильность при alpha = 0.1 [4].

При изменении alpha на лучший результат, правильность работы модели увеличилась практически до 1, что видно на рис. 4.

```
Метрика accuracy:  0.7222222222222222
Правильные результаты:  0         летучая мышь
1              бабочка
2                птица
3              человек
4              медведь
5                 слон
6              собака
7             черепаха
8               облако
9              ящерица
10               тигр
11    голова человека
12               краб
13            осьминог
14               паук
15             кролик
16               заяц
17                лев
Name: label, dtype: object
Результаты классификации:  ['летучая мышь' 'бабочка' 'птица' 'человек' 'медведь' 'слон' 'птица'
 'птица' 'птица' 'ящерица' 'тигр' 'голова человека' 'краб' 'птица' 'птица'
 'кролик' 'заяц' 'лев']
```

Рис. 2. Правильность работы модели до подбора параметров.
Fig. 2. Accuracy of model performance prior to parameter selection.

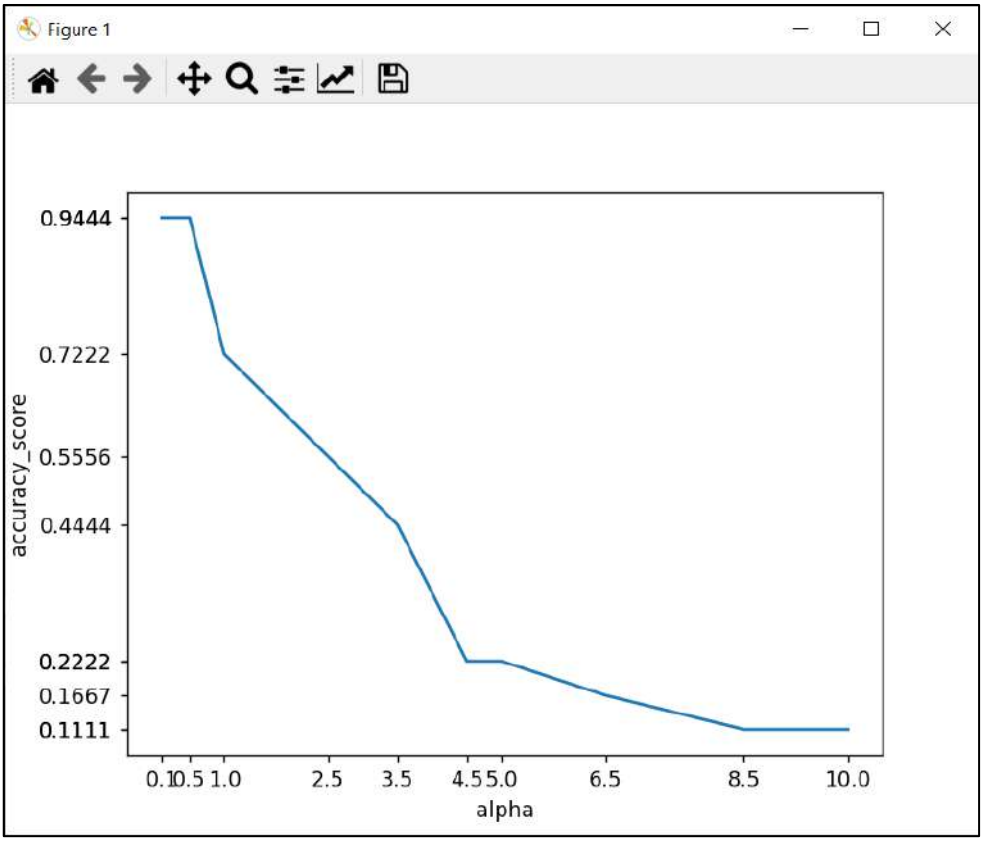


Рис. 3. Подбор параметра alpha.
Fig. 3. Selection of alpha parameter.

```
Метрика accuracy:  0.9444444444444444
Правильные результаты:  0         летучая мышь
1              бабочка
2                птица
3              человек
4              медведь
5                 слон
6              собака
7             черепаха
8                облако
9              ящерица
10                тигр
11    голова человека
12                краб
13           осьминог
14                паук
15              кролик
16                заяц
17                 лев
Name: label, dtype: object
Результаты классификации:  ['летучая мышь' 'бабочка' 'птица' 'человек' 'медведь' 'слон' 'птица'
 'черепаха' 'облако' 'ящерица' 'тигр' 'голова человека' 'краб' 'осьминог'
 'паук' 'кролик' 'заяц' 'лев']
```

Рис. 4. Результат работы модели после подбора параметров.
Fig. 4. Model result after parameter selection.

Результатом работы модели всегда будет одна из категорий и вероятность принадлежности. Поскольку наш набор данных не содержит колонки с категорией «другое», а введённый пользователем текст может не принадлежать ни одной из доступных категорий, добавим проверку. Если вероятность принадлежности к одной из доступных категорий меньше 50%, то это будет означать, что введённый текст не относится ни к одной из категорий.

```
if max(predicted_probabilities) > 0.5:
            return result
return "другое"
```

Листинг 2. Проверка точности результата.
Listing 2. Checking the accuracy of the result.

3.2 Модель категоризации ответа по содержанию

Для определения категории ответа следует использовать модель многоклассовой классификации с несколькими метками, так как ответ тестируемого может содержать слова одновременно из нескольких категорий. Бейзлайн ML-модель в таком случае будет работать очень медленно и давать низкую правильность прогноза. Наиболее простой вариант реализации – воспользоваться готовой открытой языковой моделью, такой как BERT или LLaMa с добавлением RAG-архитектуры или дообучением.

Поскольку набор данных для решаемой задачи нам тоже был собран вручную и имеет небольшие размеры, дообученная модель сильно ухудшится в качестве, так как данных слишком мало. Отправка запросов к модели по API добавлением RAG-архитектуры будет работать при низких температурах модели, что также ухудшит качество ответа, но этот вариант более релевантный в рамках ограничений нашего набора данных. В качестве LLM модели была выбрана LLaMa 3.1, у неё нет ограничения по количеству токенов, так как её можно развернуть локально. Плюсом этой модели является наличие Ollama – программа с визуальным пользовательским интерфейсом, позволяющая развернуть модель без настройки кода.

Для работы с LLaMa была создана RAG-архитектура с помощью langchain, представленная в листинге 3. При каждом запросе в модель передаётся инструкция и список категорий.

```
instructions = """Use context to answer the question. Don't make up your
own categories. If you can not select, just say about it"""
prompt_template = """Determine which category {categories} its {input}
belongs to.
You can select multiple categories. """
categories = '; '.join(meaning_dict.values())

prompt = PromptTemplate(
template=prompt_template, input_variables=["categories", "input"],
)
```

Листинг 3. Промпт для LLM.
Listing 3. Prompt for LLM.

Преимущества LLaMa:

1. Возможность получения нескольких категорий.
2. Простая настройка, быстрый fine-tuning.

Недостатки LLaMa:

1. Абстрактная форма ответа. Для последующего подсчёта категорий требуется использовать поиск по подстроке, что снижает оптимизацию программы.
2. Ошибки при переводе на английский. Модель является мультиязычной, однако внутри модели работа осуществляется на английском языке. Наша программа рассчитана на русскоязычных пользователей, соответственно ответ в модель отправляется на русском. В ходе работы возникали ошибки перевода, например модель переводила «летучая мышь» как «flying mouse», хотя правильным вариантом будет «bat».
3. Даже при низких температурах модель иногда выдаёт собственные категории.
4. Большое количество ошибок при категоризации.

Результаты классификации не самые хорошие, поэтому было принято решение самостоятельно реализовать DL-модель на keras для tensorflow [12, 14]. В качестве входных данных – набор данных размерностью 4000 строк с примерами предложений и результатами классификации по одной из 21 категории. Если данные подходят под категорию, напротив них в столбце стоит 1. Фрагмент набора данных представлен на рис. 5.

	answers	H	(H)	...	Bi	Ti	Ci
Изображение руки, сжимающей кисть другой руки.	1	0	1	...	0	0	0
Фотография ног, идущих по тропинке в лесу.	1	0	1	...	0	0	0
Человеческие силуэты, обнимающие друг друга на ...	1	0	0	...	0	0	0
Фотография человеческой головы, покрытой пеной ...	1	0	1	...	0	0	0
Человеческая спина, выпрямленная во время йоги.	1	0	1	...	0	0	0
...	...	..	...	...	..	..	..
Облака создают невероятные формы и образы.	0	0	0	...	0	0	1
Облака словно непостижимое море в небесных прос...	0	0	0	...	0	0	1
Облака плывут над горизонтом.	0	0	0	...	0	0	1
Облака создают контраст с голубым небом.	0	0	0	...	0	0	1
Облака расстилаются по небу.	0	0	0	...	0	0	1

Рис. 5. Фрагмент набора данных.
Fig. 5. Dataset fragment.

Для передачи обучающих данных в модель необходимо предварительно представить их в виде тензоров. Текстовые данные предобработали (удалили стоп-слова и пунктуацию с помощью spacy, лемматизировали с помощью трансформера от bert и разбили на токены) и векторизовали с помощью TfIdfVectorizer от scikit-learn. Поскольку столбцы категориями содержат 0 и 1, их сразу можно представить в виде вектора, необходимо только привести к нужному типу. Представление данных в виде тензоров и приведение к нужному типу показано в листинге 4 [15].

```
vectorizer = TfIdfVectorizer(binary=True)
x_train=
vectorizer.fit_transform(dataframe['answers'].astype(str)).toarray()
y_train = np.asarray(y_dataframe).astype('float32')
```

Листинг 4. Перевод в тензоры и приведение к нужному типу.
Listing 4. Conversion to tensors and reduction to the required type.

Далее создадим простую полносвязную сеть, состоящую из четырёх слоёв. Код создания представлен в листинге 5.

```
model.add(Dense(128, activation='LeakyReLU'))
model.add(LSTM(64, return_sequences=True))
model.add(GlobalAveragePooling1D())
model.add(Dense(21, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Листинг 5. Создание полносвязной сети.
Listing 5. Creating a fully connected network.

В качестве функции активации используем sigmoid для последнего слоя для многозначной классификации. В таком случае на выходе мы получим вероятность принадлежности к каждой категории в диапазоне от 0 до 1. В качестве выходных значений мы получим шансы результирующей вероятности. Эта функция очень часто используется в примерах многоклассовой классификации. Для первого слоя используем 'LeakyReLU' вместо классического relu, используемого для классификации, поскольку он помогает избежать проблемы умерших нейронов [16] Слой LSTM отвечает за обработку последовательностей, таким образом мы можем рассматривать связки слов или контекст.

Также стоит отметить, что выходной слой модели имеет 21 нейрон, что равно количеству категорий в нашем наборе данных. Другими словами, для каждого выхода модели будет свой отдельный вектор, который будет представлять каждый отдельный класс [17].

В качестве функции потерь была выбрана categorial_crossentropy, которая измеряет расстояние между истинными и предсказанными распределениями. Как понятно из названия, она также используется в задачах категоризации и определяет расстояние между распределениями вероятностей между истинным и предполагаемым значениями. Таким образом, она улучшает работу модели, минимизируя это расстояние [18].

Правильность работы данной модели представлена на графике, который изображен на рис. 6.

Существенным недостатком такой модели является ограниченность вариантами ответа. Эта проблема решается увеличением набора данных. Для сравнения работы моделей, было произведено реальное тестирование, ответы, данные пользователем, были переданы в модель. В табл. 2 представлены категории, к которым модели относили введённый текст. По ответам нашей модели можем видеть, что часто появляется категория «Внутренние органы животного». Для устранения этого «перевеса» необходимо также увеличить набор данных, дополнить примерами остальные категории.

4. Подсчёт оценки по категории «Тип переживания»

Далее результаты, полученные в процессе тестирования и работы моделей передаются в методы подсчёта результатов по каждой из категорий:

- Общая оценка личности – характеристика по каждой из детерминант. Для теста Роршаха существуют пределы «нормы» по каждой из детерминант, отклонение от нормы может дать комментарий к оценке личности. В совокупности они могут дать более полную характеристику о типе переживания и оценки личности человека, для этого необходимо сравнить значения, полученные в ходе тестирования, с показателями «нормы» [19].
- Тип переживания. По Роршаху существует 5 типов переживания: коартивный, коартированный, амбиэквальный, интраверсивный и экстраверсивный. Каждой из этих категорий соответствует своё соотношение M (показатель движения) и C (показатели по цвету) [20].
- Механизм защиты. Совокупность показателей по некоторым детерминантам может указать на конфликт. При наличии конфликта, необходимо также вычислить механизм защиты.
- Интеллектуальные возможности. Совокупность показателей по некоторым детерминантам, такими как оригинальность и ответы с наличием категории «Животное», может классифицировать интеллект как высокий, низкий или используемый не в полной мере [21].

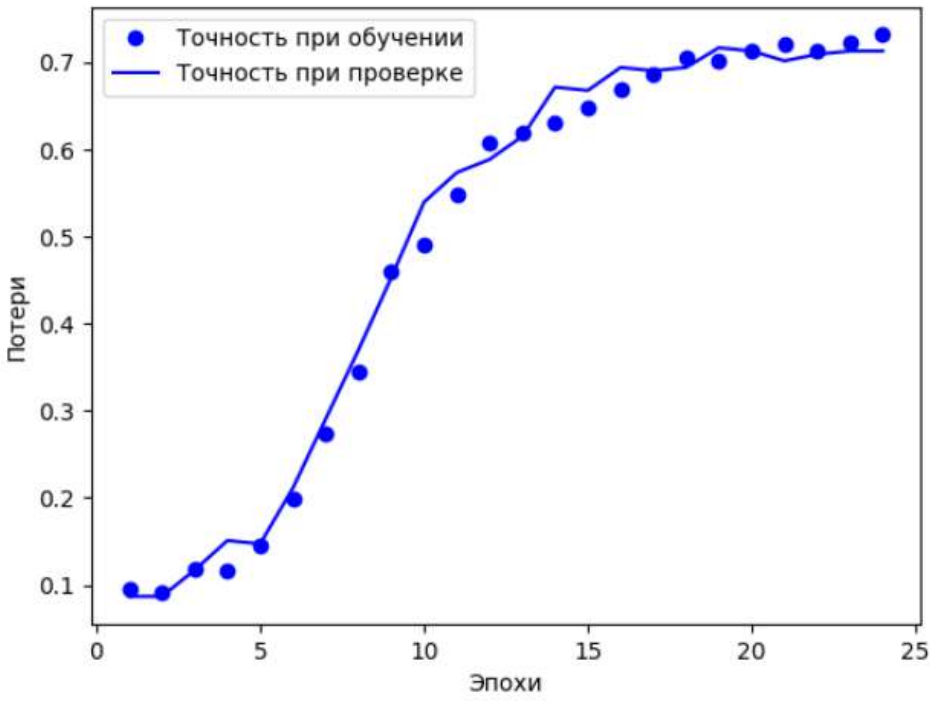


Рис. 6. График точности работы модели.
Fig. 6. Graph of model performance accuracy.

Табл. 2. Ответы моделей.
Table 2. Model's answers.

№ карточки	Ответ тестируемого	Наша модель	LLaMa 3.1
1	Чаша с горящими лепестками	Растения, Предмет, созданный человеком, Внутренние органы животного	Категория не указана
2	Летучая мышь	Животное	Животное Мифическое животное
3	Лапы паука или краба	Часть животного	Часть животного
4	Бабушкина шуба	Нереалистичная фигура человека, Внутренние органы животного	Детский рисунок
5	Бабочка	Животное	Животное, Часть животного
6	Большой контрабас или гитара	Детский рисунок	Географическая карта
7	Пузатые кролики	Внутренние органы животного, Еда	Детский рисунок
8	Птица	Внутренние органы животного, Географические карты	Животное, Часть животного, Мифическое животное
9	Лёгкие человека	Внутренние органы человека	Человек, Животное, Часть животного, Абстрактная концепция
10	Много насекомых	Часть животного	Часть животного

В качестве примера рассмотрим выставление результата по категории «Тип переживания». Блок-схема работы метода представлена на рис. 7. Функция получает на вход значения по детерминантам M, C, CF, FC, где M – кинестетические а C, CF, FC – цветовые и цвето-формовые показатели. Для диагностики типа переживания используется соотношения параметров M и sum C, где M — количество ответов с человеческими кинестезиями, SumC — количество ответов с использованием хроматического цвета, которое высчитывается по формуле (2).

$$Sum\ C = \frac{3C + 2CF + 1FC}{2} \quad (2)$$

Как видно на блок-схеме, соотношения M и sum C характерны для следующих типов переживания:

1) коартированный (0:0, 1:0, 0:1, 1:1);
2) коартивный (показатели не выше 3х с каждой стороны);

3) амбиэквальный (примерно равные 5:6, 8:8, 9: 11);
4) интраверсивный (значительно преобладает М);
5) экстратенсивный (значительно преобладает С).

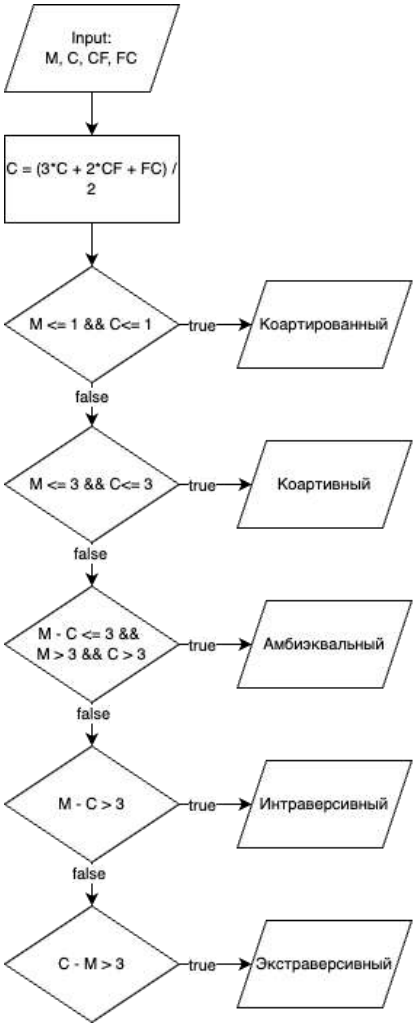


Рис. 7. Блок-схема выставления результата по категории «Тип переживания».
Fig. 7. Scoring flowchart for the "Type of Experience" category.

5. Вывод результата

После обработки результата пользователь получает индивидуальную характеристику личности по каждой из категорий, представленную на рис. 8.

6. Заключение

Разработка приложения для профессионального тестирования с возможностью ввода свободного ответа на карточку и последующей обработкой данных с применением нейронных сетей представляет собой важный шаг в области современных технологий. Этот подход позволяет значительно улучшить процесс проведения проективных методик

психологического тестирования, обеспечивая более точную и объективную интерпретацию результатов.

Созданное нами приложение требует дальнейших исследований и модернизации. В дальнейшей работе мы планируем повысить качество классификации текста по содержанию, заменив собственную модель на дообученную большую языковую модель.

Рис. 8. Результат прохождения теста в веб-интерфейсе.
Fig. 8. Test result in the web interface.

Список литературы / References

[1]. де Траубенберг Р., Роршаха Н. К. Т. практическое руководство/НК Рауш де Траубенберг //М.: Когито-Центр. – 2005. – 255 с.

[2]. Ванеян С. С. Тест Роршаха: случайности восприятия и закономерности воображения //Современная терапия в психиатрии и неврологии. – 2014. – №. 2. – С. 45-52.

[3]. Белый Б. И. Тест Роршаха. Практика и теория / Под ред. Л. Н. Собчик. — СПб.: ООО «Каскад», 2005. - 240 с.

[4]. Мартышкин А.И., Григорьева Д.Д., Серов Д.В., Сорокин Д.С. Использование нейронных сетей для анализа ответов на проективные методики психологического тестирования на примере теста Роршаха. – XXI век: итоги прошлого и проблемы настоящего плюс. 2023. Т. 12. № 3 (63). С. 42-49

[5]. Нохрина Н. А. Типы переживания личности и особенности восприятия субъектами произведений искусства //Психология. Психофизиология. – 2010. – №. 17 (193). – С. 89-92.

[6]. Кольцов Д. М. Python. Полное руководство //СПб: Издательство Наука и Техника. – 2022. – 480 с.

[7]. Хасти Т., Тибширани Р., Фридман Д. Основы статистического обучения. Интеллектуальный анализ данных, логический вывод и прогнозирование //Санкт-Петербург: ООО “Диалектика. – 2020. - 768 с.

[8]. Rish I. et al. An empirical study of the naive Bayes classifier //IJCAI 2001 workshop on empirical methods in artificial intelligence. – 2001. – Т. 3. – №. 22. – С. 41-46.

[9]. Маннинг К. Д., Рагхаван П., Шютце Х. Введение в информационный поиск, Вильямс, М., 2011 //English: Manning CD, Raghavan P., Schutze H., Introduction to Information Retrieval, Cambridge University Press New York, NY, USA. – 2008. - 504 с.

[10]. Кадурин А., Николенко С., Архангельская Е. Глубокое обучение. Погружение в мир нейронных сетей //СПб.: Питер. – 2018. – Т. 480.

[11]. Князева М. Д., Карамышева Н. С., Григорьева Д. Д. От алгоритма к программе и искусственному интеллекту. С примерами на языках Python и Prolog: учеб. пособие / Под ред. д-ра техн. наук, проф. С. А. Зинкина. – Пенза: Изд-во ПГУ, 2024. – 412 с.

[12]. Джулли А., Пал С. Библиотека Keras–инструмент глубокого обучения. Реализация нейронных сетей с помощью библиотек Theano и TensorFlow. – Litres, 2022. - 296 с.
[13]. Shukla N., Fricklas K. Machine learning with TensorFlow. – Greenwich: Manning, 2018. - 272 с.
[14]. Черняк Е. Введение в глубокое обучение/Е //Черняк–Спб: Изд-во Диалектика. – 2020. – 192 с.
[15]. Франсуа Ш. Глубокое обучение на Python. – СПб.: Питер, 2022. — 400 с.: ил. — (Серия «Библиотека программиста»).
[16]. Hunt J., Graphing with Matplotlib pyplot //Advanced Guide to Python 3 Programming. – 2019. – С. 43 - 65.
[17]. KOYUNCU, Y. Mixed Methods in Educational Sciences: A Qualitative Content Analysis of Master s Theses [Electronic resource] / Y. KOYUNCU, İ. K. YÜKSEL // Hacettepe University Journal of Education. — 2022. — Available from: https://doi.org/10.16986/huje.2022.458.
[18]. Гафаров Ф. М., Галимянов А.Ф. Искусственные нейронные сети и приложения. – Казань: Изд-во Казан. ун-та, 2018. – 121 с.
[19]. Mihura J. L. et al. The validity of individual Rorschach variables: systematic reviews and meta-analyses of the comprehensive system //Psychological bulletin. – 2013. – Т. 139. – №. 3. – С. 548.
[20]. Грязева-Добшинская В. Г., Нохрина Н. А. Тип переживания и рефлексия образов я при восприятии произведений визуального искусства //Психология. Психофизиология. – 2013. – Т. 6. – №. 4. – С. 15-23.
[21]. Бурлачук Л. Ф. Введение в проективную психологию/Леонид Фокич Бурлачук //К.: Ника-Центр. – 1997. – С. 78-79.

Информация об авторах / Information about authors

Дарья Дмитриевна ГРИГОРЬЕВА – выпускница кафедры «Вычислительная техника». Сфера научных интересов: искусственный интеллект, нейронные сети, автоматизированное тестирование, веб разработка.

Daria Dmitrievna GRIGORIEVA – graduate of the Department "Computer Science". Sphere of scientific interests: artificial intelligence, neural networks, automated testing, web development.

Даниил Валерьевич СЕРОВ – студент кафедры «Программирование». Сфера научных интересов: машинное обучение, автоматизированное тестирование, нейронные сети, бэкенд разработка.

Daniil Valeryevich SEROV – student of the Department "Programming". Research interests: machine learning, automated testing, neural networks, backend development.

Дмитрий Сергеевич СОРОКИН – студент кафедры «Программирование». Сфера научных интересов: машинное обучение, мобильная разработка, нейронные сети.

Dmitry Sergeyevich SOROKIN – student of the Department "Programming". Research interests: machine learning, mobile development, neural networks.

Алексей Иванович МАРТЫШКИН – кандидат технических наук, доцент, заведующий кафедрой «Программирование». Сфера научных интересов: обработка данных, моделирование вычислительных систем, исследование высокопроизводительных систем, распределенные вычислительные системы.

Alexey Ivanovich MARTYSHKIN – Cand. Sci. (Tech.), Associate Professor, Head of the Department "Programming". Research interests: data processing, modeling of computing systems, research of high-performance systems, distributed computing systems.

DOI: 10.15514/ISPRAS-2025-37(1)-12

Математическое моделирование почвенных процессов с использованием открытого программного обеспечения

К.Б. Кошелев, ORCID: 0000-0002-7124-3945 <k.koshelev@ispras.ru>
А.В. Кулинский, ORCID: 0009-0004-6407-054X <a.kulinsky@ispras.ru>
С.В. Стрижак, ORCID: 0000-0001-5525-5180 <s.strijhak@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В настоящее время задача построения цифровых двойников различных природных и технических объектов является актуальной задачей. В работе рассматриваются возможности открытого ПО для разработки цифровых двойников почвенных процессов. Облачная платформа для проведения научных исследований может быть спроектирована и создана на базе аппаратно-программного комплекса, куда входят такие компоненты как сервера, система хранения данных, сетевое оборудование, стек системного программного обеспечения, виртуальные машины, микросервисы и другие элементы. Облачная платформа может быть выступать основной для проектов разработки цифровых двойников. В качестве исходных данных для построения цифрового двойника почвенных полей могут выступать метеорологические данные, данные по цифровому рельефу местности, данные по физико-химическому составу почвы, данные по сельскохозяйственным культурам, синтетические данных. В статье рассматриваются возможности открытых программных комплексов ParFlow, OpenFOAM, Paraview для моделирования почвенных процессов с использованием уравнения Ричардса для однофазной среды**.** Физическое моделирование проведено для случаев задания модельных почвенных полей с заданной проницаемостью и пористостью среды. В результате расчета получены поля влагонасыщенности, гидростатического напора, скорости движения влаги. В одной из модельных задач также исследовалось влияние наличия скважины на поле давления. Базовая расчетная сетка включала в себя 288 000 расчетных ячеек. Типовые расчеты проведены на вычислительном кластере ИСП РАН. Один типовой расчет запускался на 12 ядер и выполнялся по времени около 20 минут. Визуализация результатов расчета выполнена в пакете Paraview с использованием технологии фильтров и программных скриптов на языке программирования Python.

Ключевые слова: цифровой двойник; почвенное поле; моделирование; сетка; поле; скорость; давление; влагонасыщенность; визуализация; вычислительный кластер.

Для цитирования: Кошелев К.Б., Кулинский А.В., Стрижак С.В. Математическое моделирование почвенных процессов с использованием открытого программного обеспечения. Труды ИСП РАН, том 37, вып.1, 2025 г., стр. 201–216. DOI: 10.15514/ISPRAS–2025–37(1)–12.

Благодарности: Работа выполнена при финансовой поддержке Министерства науки и высшего образования Российской Федерации (Соглашение № 075-15-2024-545 от 24 апреля 2024 года).

Mathematical Modeling of Soil Processes using Open-Source Software

K.B. Koshelev, ORCID: 0000-0002-7124-3945 <k.koshelev@ispras.ru>
A.V. Kulinsky, ORCID: 0009-0004-6407-054X <a.kulinsky@ispras.ru>
S.V. Strijhak, ORCID: 0000-0001-5525-5180 <s.strijhak@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Nowadays the task of building digital twins of various natural and technical objects is an urgent task. The paper considers the possibilities of open source software for the development of digital twins of soil processes. Cloud platform for scientific research can be designed and created on the basis of hardware and software complex, which includes such components as servers, data storage system, network equipment, system software stack, virtual machines, microservices and other elements. The cloud platform can act as the main platform for digital twin development projects. Meteorological data, data on digital terrain relief, data on physical and chemical composition of soil, data on agricultural crops, synthetic data can be used as input data for building a digital twin of soil fields. In the article the possibilities of open-source software packages ParFlow, OpenFOAM, Paraview for modeling of soil processes using Richards equation for single-phase medium are considered. Physical modeling was carried out for cases of model soil fields with given permeability and porosity of the medium. As a result of the calculation, the fields of moisture saturation, hydrostatic head, and moisture velocity were obtained. In one of the modeling problems the influence of well presence on the pressure field was also investigated. The basic calculation grid included 288,000 calculation cells. Typical calculations were performed on the high-performance cluster of ISP RAS. One typical calculation was run on 12 cores and took about 20 minutes. Visualization of the calculation results was performed in the Paraview package using filter technology and software scripts in the Python programming language.

Keywords: Digital twin; soil field; modeling; grid; field; velocity; pressure; moisture saturation; visualization; computational cluster.

For citation: Koshelev K.B., Kulinsky A.V., Strijhak S.V. Mathematical modeling of soil processes using open-source software. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 201-216 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-12.

Acknowledgements. The work was carried out with the financial support of the Ministry of Science and Higher Education of the Russian Federation (Agreement No. 075-15-2024-545 dated April 24, 2024).

1. Введение

Россия является лидером в мире по площади земель сельхозназначения, находится в первой пятерке стран по площади пашни и запасам природных ресурсов, обладает необходимыми природными условиями для обеспечения продовольственной безопасности страны.

В стратегии развития РФ сформулирована принципиальная задача перевода экономики на качественно новые принципы работы с точки зрения управления на основе больших данных как о системообразующей инфраструктуре для дальнейшего развития страны с использованием цифровых платформ в различных отраслях и, в частности, цифровых двойников технических систем и процессов, применения точного земледелия в сельском хозяйстве.

Цифровые двойники – одно из самых развивающихся направлений в области компьютерных наук, основанное на конвергенции передовых информационных технологий, включая киберфизические системы, компьютерное моделирование, Интернет вещей, большие данные, искусственный интеллект, виртуализацию вычислительных ресурсов и ряд других.

Можно выделить три направления, на решение которых направлено создание виртуальных образов, в которых хранится вся информация о состояниях физической системы – цифровых двойников почв:

1) управление процессами в растениеводстве – основа адаптивно-ландшафтной системы земледелия нового технологического уклада;
2) реализации сценарных прогнозов трансформации почвенного покрова в результате хозяйственной деятельности и улучшения качества почвы в контексте модельного предикативного управления;
3) оценка экономической эффективности возделывания с/х культур с учетом набора агротехнологий и изменения почвенно-климатических ресурсов.

В настоящее время современные подходы к реализации цифровых двойников почв и агроландшафтов лишены концептуальной основы, что затрудняет применения и понимания актуальности использования цифровых двойников в растениеводстве, что определяет актуальность данной работы.

Использование цифровых двойников в сельском хозяйстве еще не вышло на практический уровень. С одной стороны, сельскохозяйственные процессы обычно более сложны, чем промышленные процессы. Эта сложность связана не только с высокой размерностью данных, но и с тем фактом, что многие переменные, обуславливающие поведение процессов, носят стохастический характер и не поддаются управлению или контролю. Кроме того, большая площадь, на которой происходят сельскохозяйственные процессы, в сочетании с неоднородными условиями, характерными для этих территорий, требует пространственного и временного разрешения данных, что сложно технически и экономически.

В целом, концепция адаптивно-ландшафтного земледелия тесно связана с парадигмами управления растениеводством, где к стандартным производственным процессам добавляется слой информационных технологий. Пространственная и временная информация, полученная из множества различных источников, может быть интегрирована в алгоритмы машинного обучения и в нейронные сети для достижения контроля над всеми операционными аспектами производственной деятельности [1-3].

2. Концепция цифровых двойников

В настоящее время в ИСП РАН ведется разработка облачной платформы на базе аппаратных и собственных программных решений (Asperitas, Michman, Talisman, FanLight) для создания цифровых двойников почвенных полей в рамках Крупного Научного Проекта с Министерством Науки и Высшего Образования РФ. На рис. 1 представлена общая схема работы пользователя с данными, с математическими моделями и программными библиотеками в рамках созданных виртуальных машин, с тематической базой данных, с системой научной визуализации результатов расчета, с системой формирования набора данных и моделей машинного обучения моделей.

В качестве исходных данных для создания цифрового двойника почвенного поля могут выступать метеорологические данные с метеостанций и моделей регионального климата, данные по цифровому рельефу местности, данные по физико-химическому и биологическому составу почвы, данные по сельскохозяйственным культурам, синтетические данные и другие (рис. 2). На первом этапе в качестве математических моделей могут быть рассмотрены базовые модели в составе открытых программных кодов и библиотек (OpenFOAM, ParFlow, APSIMX, Delft3D, WRF-Hydro), библиотек для научной визуализации почвенных процессов и структур (Paraview, VisIt, Fiji), модели для физически-информированных нейронных сетей (DeepXDE, Nvidia Modulus, PyTorch, TensorFlow, Keras).

В основе концепции цифровых двойников заложена идея использования нейронных сетей для проведения расчетов в режиме реального времени.

Полученные синтетические данные могут быть использованы для разработки и обучения физически-информированных нейронных сетей для аппроксимации уравнения Ричардса для однофазной и многофазной среды в дальнейшей работе (рис. 2).

Рис.1. Общая схема работы с виртуальными вычислительными ресурсами.
Fig.1 General scheme of work with virtual computing resources.

Физически-информированная нейронная сеть (Physics-Informed Neural Network – PINN) состоит из трех основных блоков. Первая часть включает в себя модуль для вычисления остаточных слагаемых для дифференциальных уравнений в частных производных или относительную погрешность решения в норме L2, а также погрешности для начальных и граничных условий. Параметры для полносвязанной нейронной сети определяются путем нахождения минимума для общей функции потерь. Входы для нейронной сети преобразуются в соответствующие выходы. Вторая часть это полносвязанная нейронная сеть с физическими данными, которая берет выходные поля скоростей и вычисляет их производные, используя исходные уравнения, например, для уравнений неразрывности и количества движения при решении задач механики жидкости. Также оцениваются граничные и начальные условия, данные наблюдений из эксперимента. Последним шагом является механизм формирования обратной связи, который минимизирует функцию потерь, используя заданный оптимизатор (Adam, L-BFGS-B), в соответствии с некоторой скоростью обучения, чтобы получить оптимальные параметры для нейронной сети.

3. Математические модели

При изучение почвенных процессов различают три характерных масштаба. Это уровень микропорового пространства, уровень почвенного профиля, уровень агроландшафта.

Для генерации синтетических данных с целью разработки PNN могут быть использованы численные методы решения уравнений для гидрологии с переносом веществ на трех масштабных уровнях с применением адаптивной пространственно-временной расчетной сетки:

1) микроуровень – пористая среда (на основе томограмографической структуры почвы и текстуры поверхности), моделирование многофазного потока (обобщенный закон Дарси) в пористой среде с капиллярными явлениями;
2) уровень почвенного профиля – моделирование потока и диффузии воды в почве решением уравнения Ричардса;
3) ландшафтный уровень (на основе цифровой модели рельефа и карт свойств) – решение уравнений по теории мелкой воды с впитыванием с поверхности [10-12].

В данной работе рассматриваются возможности двух открытых программных комплексов (ParFlow, OpenFOAM) для изучения уровня почвенного профиля, уровня микропорового пространства. Оба программных комплекса доступны в виде исходных файлов, которые можно скачать из Интернета и откомпилировать с нужными библиотеками [4-7].

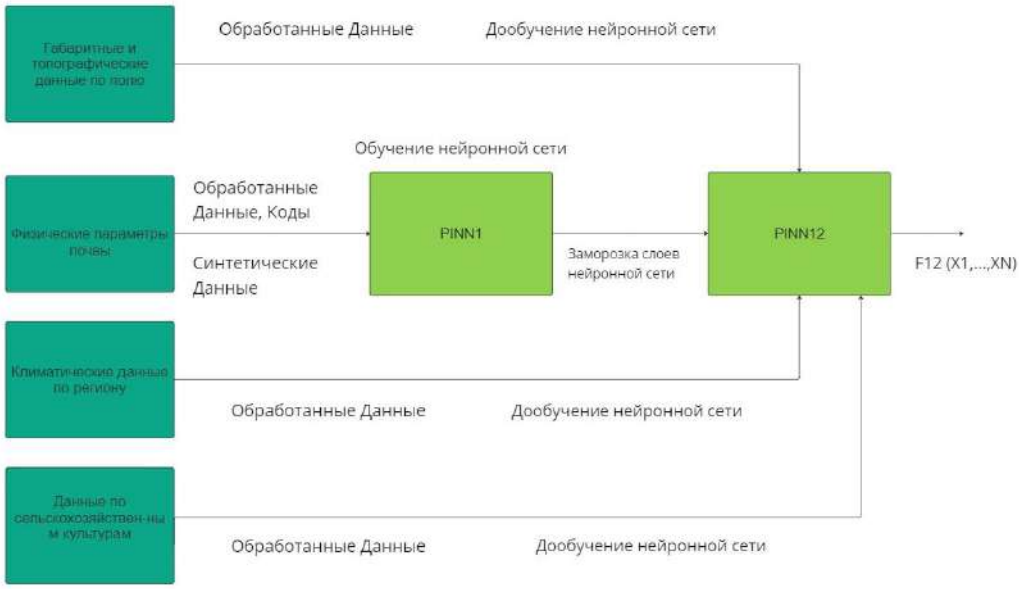


Рис. 2. Общая схема работы с данными и PINN.
Fig. 2. General scheme of work with data and PINN.

3.1 Программный комплекс ParFlow

ParFlow – открытый программный комплекс, предназначенный для моделирования гидрологических процессов с учетом взаимного влияния поверхностных и подземных вод.

Данный программный пакет позволяет эффективно учитывать сложные трехмерные пространственные структуры – топографию, различные геологические и гидрологические характеристики, границы водоразделов. Важной особенностью пакета ParFlow является возможность параллельных вычислений с использованием стандарта MPI. Все это позволяет проводить моделирование конечно-разностным методом на участках с значительными пространственными масштабами. В пакете ParFlow поддерживается работа с форматами вывода данных SILO, NETCDF, VTK. Программная реализация ParFlow выполнена на языках программирования Си и Fortran [4-5]. Пакет ParFlow позволяет использовать несколько решателей – стационарное движение грунтовых вод, влагоперенос в ненасыщенной зоне, совместное движение подземных и поверхностных вод с учетом влияния климата, растительности и других факторов.

3.1.1 Модель стационарного движения грунтовых вод

Для нахождения стационарного, полностью насыщенного потока грунтовых вод используется уравнение [9-10]:

$$-\nabla(\boldsymbol{K}\nabla H) = Q, \tag{1}$$

где $\boldsymbol{K}$ – тензор гидравлической проводимости, H – гидростатический напор, Q – пространственно-распределенный источник/сток (например, для учета влияния скважин).

3.1.2 Модель влагопереноса в ненасыщенной зоне

Уравнение Ричардса, описывающее влагоперенос в ненасыщенной зоне, используется в форме [10-11]:

$$S(p)S_s\frac{\partial p}{\partial t} - \frac{\partial(S(p)\rho(p)\phi)}{\partial t} - \nabla\cdot\left(\boldsymbol{K}(p)\rho(p)(\nabla p - \rho(p)\vec{g})\right) = Q \tag{2}$$

где p – напор (пьезометрическая высота), $S(p)$ – функция водонасыщения от напора, S_s - коэффициент упругой ёмкости пласта, t – время, ϕ – пористость среды, $\boldsymbol{K}(p)$ – тензор гидравлической проводимости, Q – источник/сток.

Граничные условия могут быть записаны как:

$$p = p_D \tag{3}$$

или

$$\boldsymbol{K}(p)\nabla p\cdot\boldsymbol{n} = g_N \tag{4}$$

где p_D – заданный напор, $\boldsymbol{n}$ – нормаль к границе, g_N – заданный поток.

Начальные условия представляются в виде

$$p = p^0 \tag{5}$$

Тензор гидравлической проводимости рассчитывается по формуле:

$$\boldsymbol{K}(p) = \frac{\bar{k}k_r(p)}{\mu} \tag{6}$$

Здесь $\bar{k}$ – тензор абсолютной проницаемости, $k_r(p)$ – относительная проницаемость, обычно вычисляемая по формуле Ван Генухтена, μ – вязкость воды.

3.1.3 Модель движения поверхностных вод

ParFlow позволяет моделировать совместное движение поверхностных и грунтовых вод. Для движения поверхностных вод используется уравнение кинематической волны:

$$\frac{\partial\psi_S}{\partial t} = \nabla\cdot(\vec{v}\psi_S) + q_r(x) + q_e(x) \tag{7}$$

Здесь ψ_S – глубина, $\vec{v}$ – осредненная по глубине скорость, $q_r(x)$ – боковая приточность/сток, $q_e(x)$ – поток, реализующий взаимодействие с грунтовыми водами.

Формула Маннинга используется для реализации уравнений движения:

$$v_x = -\frac{\sqrt{S_{f,x}}}{n}\psi_s^{2/3}, v_y = -\frac{\sqrt{S_{f,y}}}{n}\psi_s^{2/3} \tag{8}$$

Здесь S_f – гидравлический уклон, в данной модели принимаемый равным наклону уровня поверхности земли, n – коэффициент шероховатости.

На границе вода-дно задается условия непрерывности давления в виде:

$$p = \psi_S = \psi \tag{9}$$

Окончательно, уравнение (7) можно записать как:

$$-\boldsymbol{K}(\psi)\nabla\psi\cdot\boldsymbol{n} = \frac{\partial\|\psi,0\|}{\partial t} - \nabla\cdot(\vec{v}\|\psi,0\|) - q_r(x) \tag{10}$$

где $\|\psi,0\|$ – оператор максимума величин ψ и 0.

Уравнения движения для уравнения (10) используются в виде:

$$v_x = -\frac{S_{f,x}}{n\sqrt{\overline{S_f}}}\psi^{2/3}, v_y = -\frac{S_{f,y}}{n\sqrt{\overline{S_f}}}\psi^{2/3} \quad (11)$$

где $\overline{S_f}$ – абсолютное значение гидравлического уклона.

3.1.4 Модель многофазных потоков

Модель многофазного течения основана на законе Дарси. Количество фаз ν может быть от 1 до 3 (вода, нефть, газ). Номер фазы i принимает значения от 0 до $\nu - 1$.
Система уравнений движения многофазных сред принимается в виде:

$$\vec{V_i} + \lambda_i \cdot (\nabla p_i - \rho_i \vec{g}) = 0 \quad (12)$$

$$\lambda_i = \frac{\bar{k} k_{ri}}{\mu_i} \quad (13)$$

Здесь $\vec{V_i}$ – вектор скорости Дарси, p_i – давление, ρ_i – плотность, $\bar{k}$ – абсолютная проницаемость, k_{ri} – относительная проницаемость, μ_i – вязкость.
Уравнения неразрывности для $i = 1..\nu - 1$:

$$\frac{\partial(\phi S_i)}{\partial t} + \nabla \cdot \left(\frac{\lambda_i}{\lambda_T}\overrightarrow{V_T} + \sum_{j\neq i}\frac{\lambda_i\lambda_j}{\lambda_T}(\rho_i - \rho_j)\vec{g}\right) + \sum_{j\neq i}\nabla \cdot \frac{\lambda_i\lambda_j}{\lambda_T}\nabla p_{ji} - Q_i = 0 \quad (14)$$

Здесь ϕ – пористость, Q_i – источник/сток.
Для $i = 0$:

$$\sum_i S_i = 1 \quad (15)$$

Величины $\overrightarrow{V_T}$ и λ_T определяются как:

$$\overrightarrow{V_T} = \sum_i \vec{V_i} \quad (16)$$

$$\lambda_T = \sum_i \lambda_i \quad (17)$$

Капиллярное давление между фазой i и фазой j вычисляется по формуле:

$$p_{ji} = p_{j0} - p_{i0} \quad (18)$$

Уравнение для p_0 используется в виде:

$$-\sum_i \{\nabla \cdot \lambda_i(\nabla(p_0 + p_{i0}) - \rho_i \vec{g}) + Q_i\} = 0 \quad (19)$$

$$p_i = p_0 + p_{i0}, i = 1..\nu - 1 \quad (20)$$

Замыкающие соотношения:

$$p_{i0} = p_{i0}(S_0), i = 1..\nu - 1 \quad (21)$$

3.1.5 Модель переноса загрязнителей

Транспорт загрязнителей в многофазной среде определяется следующим уравнением:

$$(\phi + (1-\phi)\rho_s K_{d;j})\frac{\partial c_{ij}}{\partial t} + \nabla \cdot \left(c_{ij}\vec{V_i}\right) =$$
$$-(\phi + (1-\phi)\rho_s K_{d;j})\lambda_i c_{ij} + \sum_k^{n_I}\gamma_k^{I;i}\chi\Omega_k^I(c_{ij} - \bar{c}_{ij}^k) + \sum_k^{n_E}\gamma_k^{E;i}\chi\Omega_k^E c_{ij} \quad (22)$$

Здесь c_{ij} – концентрация j-го компонента в i-ой фазе, ρ_s – плотность твердой среды, $K_{d;j}$ – коэффициент модели мгновенной адсорбции, n_I – количество нагнетательных скважин, $\gamma_k^{I;i}$ – скорость подачи, Ω_k^I – область нагнетательных скважин, , n_E – количество добывающих скважин, $\gamma_k^{E;i}$- скорость извлечения, Ω_k^E – область добывающих скважин.
Оператор χ определяется формулой

$$\chi A(x) = \begin{cases} 1, \text{если } x \in A \\ 0, \text{если } x \notin A \end{cases} \quad (23)$$

где A – область скважины, x – точка пространства.

3.1.6 Модель водного баланса

ParFlow может вычислять водный баланс для потоков подземных и поверхностных вод с учетом модели поверхности земли clm. Уравнение водного баланса записывается в форме:

$$\frac{\Delta(Vol_{subsurface}+Vol_{surface})}{\Delta t} = Q_{overland} + Q_{evapotranspiration} + Q_{sourcesink} \quad (24)$$

Объем подземных вод рассчитывается по формуле:

$$Vol_{subsurface} = \sum_{\Omega}(S(\psi)S_s\psi\Delta x\Delta y\Delta z + S(\psi)\phi\Delta x\Delta y\Delta z) \quad (25)$$

Объем поверхностных вод рассчитывается по формуле:

$$Vol_{surface} = \sum_{\Gamma}\psi\,\Delta x\Delta y \quad (26)$$

Поток поверхностных вод $Q_{overland}$ рассчитывается в результате решения уравнения кинематической волны. Поток вследствии эвапотранспирации $Q_{evapotranspiration}$ рассчитывается в модели поверхности земли clm. Другие источники, стоки $Q_{sourcesink}$ задаются пользователем.

3.2 Библиотека porousmultiphaseFoam

Библиотека porousmultiphaseFoam реализована в программном комплексе OpenFOAM-v2306. Структура библиотеки porousmultiphaseFoam, представлена на рис. 3.

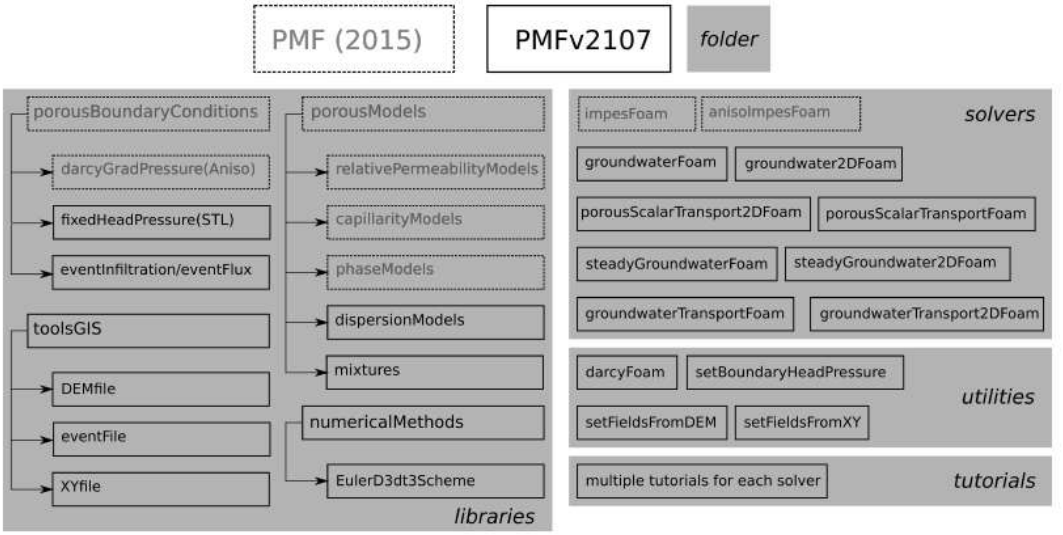


Рис. 3. Общая схема структуры библиотеки porousmultiphaseFoam.
Fig. 3. General scheme of the porousmultiphaseFoam library structure.

К основным возможностям комплекса можно отнести [6-8]:

1) набор решателей
   - groundwaterFoam,
   - stationaryGroundwaterFoam,
   - groundwater2DFoam,
   - stationaryGroundwater2DFoam,
   - porousScalarTransportFoam,
   - porousScalarTransport2DFoam,
   - groundwaterTransportFoam,
   - groundwaterTransport2DFoam,

предназначенных для моделирования потоков подземных вод, в том числе для совместного течения воды и переноса растворенных веществ;

2) улучшенные численные методы для задач с сильными нелинейностями;
3) библиотеки/утилиты для предварительной обработки входных данных (для задания информации о ГИС – toolsGIS; модуль для задания форсированных слагаемых в уравнениях, зависящих от времени для описания гетерогенной инфильтрации, локального введение трассирующих веществ);
4) модель учета влияния пассивного или связанного транспорта скаляра в решателях для моделирования грунтовых вод, поддерживающими любое количество веществ;
5) специальные граничные условия для пористых сред (porousBoundaryConditions).

Все решатели были проверены на нескольких (не)насыщенных конфигурационных примерах путем сравнения с результатами, полученными хорошо известным проверенным конечно-элементным кодом [6-8].

Данный ПК “porousMultiphaseFoam” может быть использован для генерации синтетических данных в задачах моделирования течения в пористых почвенных средах с целью дальнейшего обучения на этих данных физически-информированных нейронных сетей.

В решателе groundwaterFoam реализована математическая модель на базе системы уравнений (1-6). Для решения уравнения (2) применяется неявный метод по времени и итерационный метод Пикара.

4. *Примеры решения модельных задач*

4.1 Пример решения уравнения Ричардса для модельного поля

Рассмотрим пример в программном комплексе ParFlow для моделирования влагопереноса в почвенным поле с заданными свойствами. Размеры поля: 180м по OX, 150м по OY, 8м по OZ. Кол-во элементов разбиения соответственно: 18, 15 и 8.

Рассмотрим исходный файл в формате TCL для запуска в ParFlow. В исходном файле с именем “default_richards.tcl” производится 10 итераций решения на временном промежутке от 0.0 (TimingInfo.StartTime) до 0.010 (TimingInfo.StopTime) секунд с шагом в 0.001с.

Ниже приведены настроечные параметры для расчета:

```
pfset TimingInfo.BaseUnit           1.0
pfset TimingInfo.StartCount         0
pfset TimingInfo.StartTime          0.0
pfset TimingInfo.StopTime           0.010
pfset TimingInfo.DumpInterval       -1
pfset TimeStep.Type                 Constant
pfset TimeStep.Value                0.001
```

Параметр TimingInfo.DumpInterval означает время, когда будут генерироваться файлы вывода и записываться результаты расчета. В данном случае параметр равен -1, что означает, что решатель сам будет выбирать, когда делается запись. Также стоит обратить внимание на временные циклы и настроечные параметры:

```
pfset Cycle.Names                           constant
pfset Cycle.constant.Names                  "alltime"
pfset Cycle.constant.alltime.Length         1
pfset Cycle.constant.Repeat                 -1
```

В нашем случае имеется цикл длиной в 1 секунду (Cycle.constant.alltime.length), и он бесконечно повторяется (Cycle.constant.Repeat -1). В основном данный параметр используется для задания погодных условий, циклично друг друга сменяющих.

Также для решателей задаётся ряд параметров:

```
pfset Solver                                          Richards
pfset Solver.MaxIter                                  5
pfset Solver.Nonlinear.MaxIter                        10
pfset Solver.Nonlinear.ResidualTol                    1e-9
pfset Solver.Nonlinear.EtaChoice                      EtaConstant
pfset Solver.Nonlinear.EtaValue                       1e-5
pfset Solver.Nonlinear.UseJacobian                    True
pfset Solver.Nonlinear.DerivativeEpsilon              1e-2
pfset Solver.Linear.KrylovDimension                   10
pfset Solver.Linear.Preconditioner                    MGSemi
pfset Solver.Linear.Preconditioner.MGSemi.MaxIter     1
pfset Solver.Linear.Preconditioner.MGSemi.MaxLevels   100
```

Для дальнейшей передачи результатов расчета в Paraview требуется сохранять вывод в формате VTK, для чего использованы следующие строки (пример распределения давления представлен на рис. 4):

```
foreach i "00000 00001 00002 00003 00004 00005" {
    set Pdat [pfload -pfb default_richards_wells.out.press.$i.pfb]
    set Sdat [pfload -pfb default_richards_wells.out.satur.$i.pfb]
    pfvtksave $Pdat -vtk "default_richards_wells.out.press.$i.vtk"
                    -var "Pressure"
    pfvtksave $Sdat -vtk "default_richards_wells.out.satur.$i.vtk"
                    -var "Saturation"
}
```

Здесь в качестве переменных Pdat и Sdat представлены файлы формата .pfb (являющиеся стандартными файлами вывода в ParFlow). Затем используем команду pfvtksave пакета PFTools, для перевода .pfb файлов в формат VTK, где –var – желаемое название для данной переменной в результирующем файле.

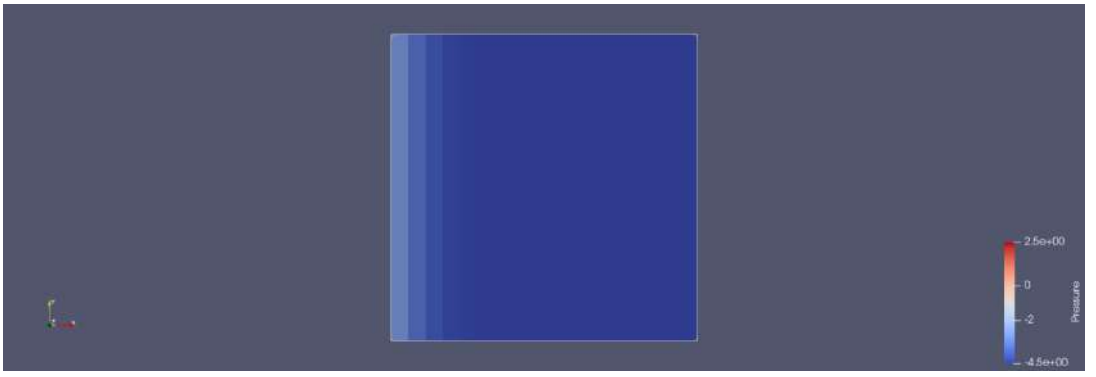


Рис. 4. Поле давления в Paraview.
Fig. 4. Pressure field in Paraview.

4.2 Пример решения уравнения Ричардса для модельного поля со скважиной

Рассмотрим пример в программном комплексе ParFlow для моделирования влагопереноса в почвенным поле со скважиной с заданными свойствами. Размеры поля: 78.8889м по OX, 116.667м по OY, 8м по OZ. Границы: Xmin = -10м; Xmax = 78.8889м; Ymin = 10м; Ymax = 116.667м; Zmin = 1м; Zmax = 9м. Количество элементов разбиения соответственно: 10, 10, 8.

Способ задачи скважины представлен следующей последовательностью команд:

```
pfset Wells.Names                                     "pumping_well"
pfset Wells.pumping_well.InputType                    Vertical
pfset Wells.pumping_well.Action                       Extraction
```

```
pfset Wells.pumping_well.Type                          Pressure
pfset Wells.pumping_well.X                             0
pfset Wells.pumping_well.Y                             80
pfset Wells.pumping_well.ZUpper                        3.0
pfset Wells.pumping_well.ZLower                        2.00
pfset Wells.pumping_well.Method                        Standard
pfset Wells.pumping_well.Cycle                         "constant"
pfset Wells.pumping_well.alltime.Pressure.Value        0.5
pfset Wells.pumping_well.alltime.Saturation.water.Value 1.0
```

Соответственно устанавливаются геометрические характеристики скважины (расположение в пространстве и её размер) и, как можно заметить, скважина не сквозная и работает на выкачивание, и её физические параметры – давление и насыщение.

Изначально данная задача предполагает временные промежутки и количество итераций, аналогичные предыдущей. После проведения расчета и перевода результатов в VTK на последнем шаге решения для давления и для насыщения появится картина, показанная на рис. 5. Если же требуется увеличить временной диапазон решения, то следует провести некоторые изменения.

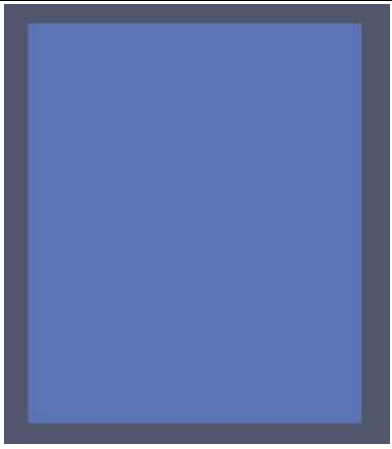
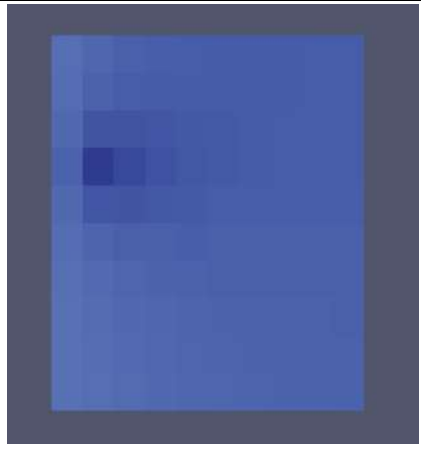
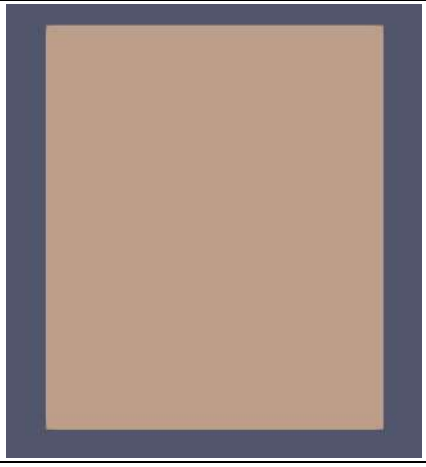
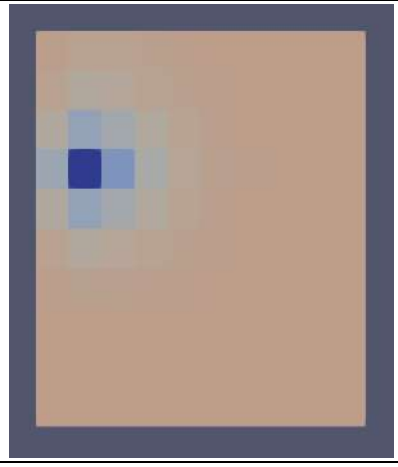
	Шаг 0	Шаг 5
Давление		
Влаго-насыщение		

Рис. 5. Поле давления и влагонасыщения в Paraview
Fig. 5. Pressure field and sarturation in Paraview

4.2.1 Временные параметры

Используем следующее настроечные параметры:

```
pfset TimingInfo.BaseUnit                                    1.0
pfset TimingInfo.StartCount                                  0
pfset TimingInfo.StartTime                                   0.0
pfset TimingInfo.StopTime                                    5.0
pfset TimingInfo.DumpInterval                                0.025
pfset TimeStep.Type                                          Constant
pfset TimeStep.Value                                         0.025
```

Для начала изменим временной отрезок (перенесем конец расчета с 0.010с до 5.0с). Увеличим шаг с 0.001с до 0.025с и установим интервал записи равный 0.025с (вывод результатов на каждом шаге решения).

4.2.2 Параметры цикла

Используем следующее настроечные параметры:

```
pfset Solver                                                 Richards
pfset Solver.MaxIter                                         20
pfset Solver.Nonlinear.MaxIter                               20
```

Так как изначально период цикла был равен одной секунде, то мы бы не смогли просчитать временной промежуток от 1с до 5с. Соответственно, изменяется параметр Cycles.constant.alltime.length.

4.2.3 Параметры решателя

Так как мы установили Dumping Interval равным 0.025с, то у нас при расчете возникнет будет 20 шагов, соответственно изменим временные параметры решателей:

Также данные операции откликнутся и на перевод файлов в формат VTK (нужно увеличить количество преобразуемых файлов):

```
foreach i "00000 00001 00002 00003 00004 00005 00006 00007 00008
           00009 00010 00011 00012 00013 00014 00015 00016 00017
           00018 00019 00020" {
    set Pdat [pfload -pfb default_richards_wells.out.press.$i.pfb]
    set Sdat [pfload -pfb default_richards_wells.out.satur.$i.pfb]
    pfvtksave $Pdat -vtk "default_richards_wells.out.press.$i.vtk"
                    -var "Pressure"
    pfvtksave $Sdat -vtk "default_richards_wells.out.satur.$i.vtk"
                    -var "Saturation"
}
```

Сравнительный рисунок изменения во времени давления и влагонасыщенности представлен на рис. 6.

Также расчет для данной задачи был выполнен на 8-12 ядрах сервера на вычислительном кластере ИСП РАН. Время расчета составило не более 20 минут.

4.3 Пример решения уравнения Ричардса для модельной задачи с использованием решателя groundwaterFoam

4.3.1 Постановка задачи

Рассматривалась возможность использования решателя groundwaterFoam в составе библиотеки porousmultiphaseFoam для решения нелинейного уравнения Ричардса. Расчетная область с заданной поверхностной топографией имела размеры: 224.4 x 113.2 x 8.5 м.

Количество расчетных ячеек задавалось равным 121x61x10 (рис. 7). Для решения краевой модельной задачи (real-field1) задавались начальные и граничные условия для величин гидростатического напора и скорости. На всех границах задавалось условие Неймана. Нелинейное уравнение Ричардса решалось с использованием метода итераций Пикара. Максимальное количество итераций задавалось равным maxIter=10. Шаг по времени для интегрирования уравнения выбирался равным deltaT=86400 секунд.

Исходные данные для расчетного примера представлены в табл. 1.

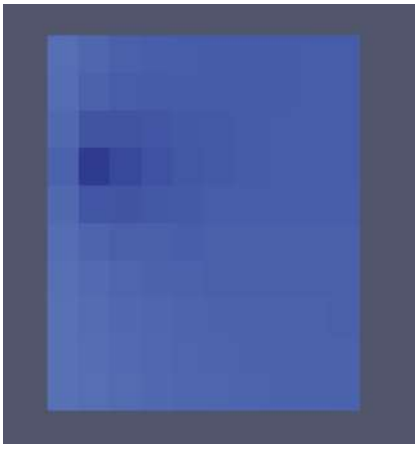
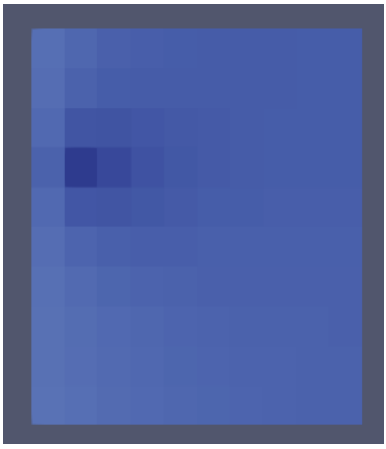
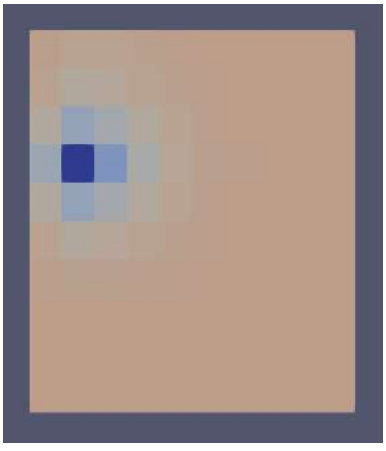
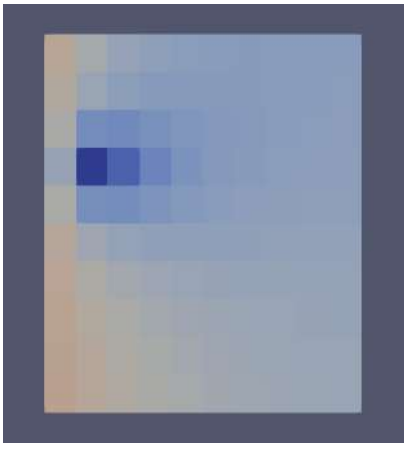
	Шаг 5 предыдущего расчета	Шаг 20 настоящего расчета
Давление		
Влаго-насыщение		

Рис. 6. Поле давления и влагонасыщения для случая со скважиной в Paraview
Fig. 6. Pressure field and sarturation with case with well in Paraview

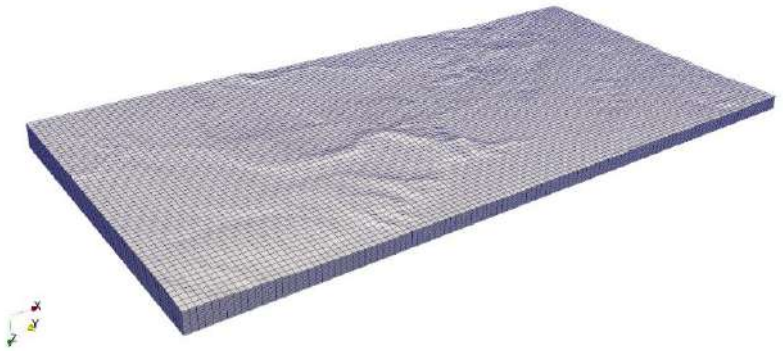


Рис. 7. Расчетная область и сетка.
Fig. 7. Numerical domain and grid.

Табл. 1. Исходные данные для примера "real-field1".
Table 1. The source data for the "real-field1" example.

viscosity	μ	1.10^{-3}	[Pa.s]
density	ρ	100	[kg.m^{-3}]
Van Genuchten coefficients	m	0.3007	[-]
	α	13.0	[m^{-1}]
Permeability	K	7.10^{-12}	[m^2]
Kinematic porosity	ε	0.27	[-]
Specific storage	S_S	0.001	[-]

(a) Flow properties

Total porosity	ε_{total}	0.30	[-]
Tortuosity	τ	1	[-]
Molecular diffusivity	D_m	1.10^{-9}	[m^2.s^{-1}]
Dispersivity coefficient	α_L	1.0	[m]
	α_T	0.2	[m]
Volume partitioning coefficient	K_d	5.10^{-5}	[m^3.kg^{-1}]
Radioactive decay coefficient	λ	1.10^{-9}	[s^{-1}]

(b) Transport properties

4.3.2 Визуализация результатов расчета в Paraview

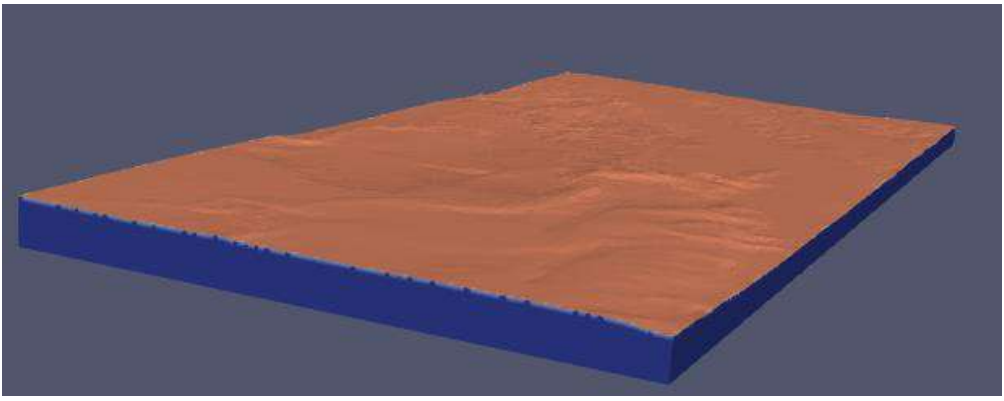
Расчет был проведен для 20 временных срезов для модели поля с заданными почвенными свойствами. В результате вычислений получены значения нескольких переменных (таких, как влагосодержание, гидростатический напор, скорость и т.д.).

Распределение влагосодержания в момент времени t=0 представлено на рис. 8a, а в момент времени 172 800 секунд - на рис. 8b.

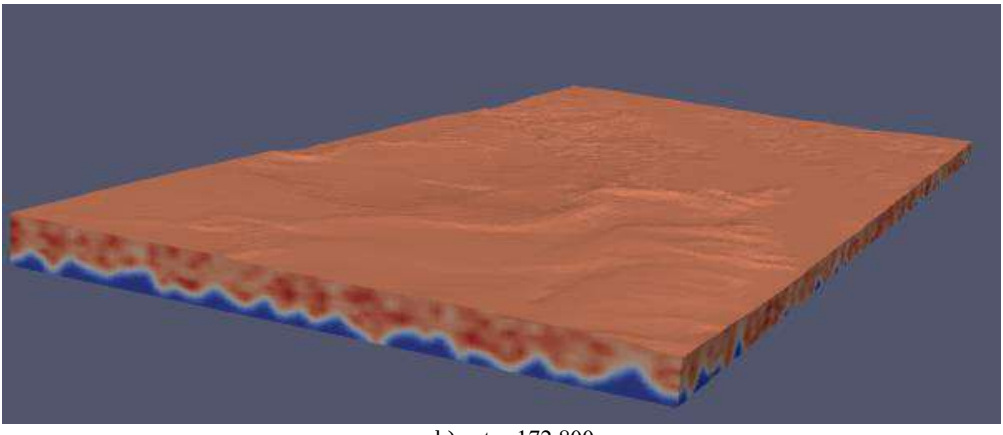
Расчет показал, что процесс распространения влаги завершился за 2.5 календарных суток. Для изучения возможностей API Paraview дополнительно были подготовлены скрипты на языке программирования Python для автоматизации процесса вывода данных в различных сечениях. Была выполнена реализация скрипта для отображения графика, выбранной переменной вдоль заданной траектории, а также скрипта для определения параметров статистики по заданному сечению, ограниченному контуром.

Заключение

Проведенное исследование позволило сделать вывод о возможном использовании программных комплексов ParFlow, OpenFOAM (porousmultiphaseFoam, poreFoam) для изучения процессов в почвенном профиле и микропоровом пространстве с заданными свойствами.



a) t = 0 с.



b) t = 172 800 с.

Рис. 8. Поле влагонасыщения для случая со скважиной в Paraview.
Fig. 8. Pressure field and sarturation with case with well in Paraview.

Список литературы / References

[1]. VanDerHorn E., Mahadevan S., Digital Twin: Generalization, characterization and implementation, Decision Support Systems, Volume 145, 2021, 113524.
[2]. Ariesen-Verschuur N., Verdouw C., Tekinerdogan B., Digital Twins in greenhouse horticulture: A review, Computers and Electronics in Agriculture, Volume 199, 2022, 107183.
[3]. Zeng Y and Su Z. Digital twin approach for the soil-plant-atmosphere continuum: think big, model small. Front Sci. 2024, 2:1376950.
[4]. Condon L. E., Maxwell R. M. Implementation of a linear optimization water allocation algorithm into a fully integrated physical hydrology model. Advances in Water Resources, 2013, 60:135–147.
[5]. Condon L. E., Maxwell R. M. Feedbacks between managed irrigation and water availability: diagnosing temporal and spatial patterns using an integrated hydrologic model. Water Resources Research, 2014, 50:2600–2616.
[6]. Horgue P., Soulaine C., Franc J., Guibert R., Debenest G. An open-source toolbox for multiphase flow in porous media. Computer Physics Communications, 187:217–226, 2015.

[7]. Horgue P., Renard F., Gerlero G.S., Guibert R., Debenest G. porousMultiphaseFoam v2107: An open-source tool for modeling saturated/unsaturated water flows and solute transfers at watershed scale, Computer Physics Communications, Volume 273, 2022, 108278.
[8]. Romanova D., Strijhak S., Koshelev K., Kraposhin M. Modeling the Reservoir Flooding Problem Using the Extended Version of the PorousMultiphaseFoam Library. 2019 Ivannikov Ispras Open Conference (ISPRAS). 05-06 December 2019, DOI: 10.1109/ISPRAS47671.2019.00025.
[9]. Пачепский Я.А. Математические модели процессов в мелиорируемых почвах. Изд-во Моск.ун-та, 1992.85 с.
[10]. Шеин Е.В. Курс физики почв. Изд-во Моск.ун-та, 2005. – 432 с.
[11]. Шеин Е.В., Рыжова И.М. (2016). Математическое моделирование в почвоведении. Учебник. – И.: Изд-во ИП «Маракушев» , 2016. – 400 с.
[12]. Куликовский А.Г., Погорелов Н.В., Семенов А.Ю. Математические вопросы численного решения гиперболических систем уравнений. Изд. ФИЗМАТЛИТ. 2001. 608 с.

Информация об авторах / Information about authors

Константин Борисович КОШЕЛЕВ – кандидат физико-математических наук, доцент, старший научный сотрудник Института системного программирования им. В.П. Иванникова РАН с 2016 года. Сфера научных интересов: вычислительная гидродинамика, гидрология, машинное обучение.

Konstantin Borisovich KOSHELEV – Cand. Sci. (Phys.-Math.), associate professor, senior researcher at the Ivannikov Institute for System Programming of the RAS since 2016. Research interests: computational fluid dynamics, hydrology, machine learning.

Андрей Всеволодович КУЛИНСКИЙ – старший лаборант Института системного программирования им. В.П. Иванникова РАН с 2024 года. Сфера научных интересов: машинное обучение, глубокое обучение, математическое моделирование, большие данные, научная визуализация.

Andrei Vsevolodovich KULINSKII – senior laboratory assistant of the Ivannikov Institute for System Programming of the RAS since 2024. Research interests: machine learning, deep learning, mathematical modeling, big data, scientific visualization.

Сергей Владимирович СТРИЖАК – кандидат технических наук, ведущий инженер Института системного программирования им. В.П. Иванникова РАН с 2009 года. Сфера научных интересов: вычислительная гидродинамика, многофазные течения, турбулентность, ветроэнергетика, машинное обучение, параллельные вычислительные системы.

Sergei Vladimirovich STRIJHAK – Cand. Sci. (Tech.), leading engineer of the Ivannikov Institute for System Programming of the RAS since 2009. Research interests: computational fluid dynamics, multiphase flows, turbulence, wind energy, machine learning, parallel computing.

DOI: 10.15514/ISPRAS-2025-37(1)-13

Модификация метода погруженных границ LS-STAG для моделирования течений неньютоновских вязких жидкостей

[1] *И.К. Марчевский, ORCID: 0000-0003-4899-4828 <iliamarchevsky@mail.ru>*
[2] *В.В. Пузикова, ORCID: 0000-0003-0712-4519 <v.puzikova@yadro.com>*

[1] *Московский государственный технический университет им. Н.Э. Баумана, 105005, Россия, г. Москва, ул. 2-я Бауманская, д. 5.*
[2] *YADRO, 123022, Россия, г. Москва, ул. Рочдельская, д. 15, с. 15.*

Аннотация. Разработана модификация метода погруженных границ LS-STAG с функциями уровня для моделирования течений неньютоновских вязких жидкостей – жидкостей, для которых вязкость в каждой точке в любой момент времени полностью определяется интенсивностью тензора скоростей деформации. Ранее была разработана модификация данного метода для другого класса неньютоновских жидкостей (вязкоупругих жидкостей), показавшая высокую точность даже для течений, которые характеризуются высокими значениями числа Вайсенберга. Поэтому представляет интерес обобщить метод LS-STAG и для неньютоновских вязких жидкостей. Отметим, что несмотря на то, что метод LS-STAG может успешно применяться для моделирования течений с подвижными погруженными границами, в данной работе внимание сфокусировано только на течениях с неподвижными границами. Построенный численный метод может использоваться как для вязкопластичных жидкостей, так и для обобщенных ньютоновских жидкостей, не обладающих пределом текучести. Для вязкопластичных жидкостей рассмотрены модели Офоли-Моргана-Штеффе, Мизрахи-Берка, Кассо и Гершеля-Балкли, используемые с моделями регуляризации Берковьера-Энгельмана и Папанастасио, а для жидкостей, не обладающих пределом текучести, модели Эллиса, Кросса, Карро, Язуды (Карро-Язуды), а также степенная модель Оствальда-де Веле. Для верификации разработанного и реализованного в авторском программном комплексе численного метода использовалась хорошо исследованная задача об обтекании неподвижного кругового профиля потоком степенной жидкости при различных значениях числа Рейнольдса и индекса течения. Полученные результаты хорошо согласуются с известными в литературе расчетными данными других исследователей. В дальнейшем планируется обобщить разработанную модификацию метода для расчета неньютоновских вязких жидкостей на случай подвижных погруженных границ.

Ключевые слова: метод погруженных границ; метод LS-STAG; неньютоновская вязкая жидкость; степенная модель; профиль; функция уровня.

Для цитирования: Марчевский И.К., Пузикова В.В. Модификация метода погруженных границ LS-STAG для моделирования течений неньютоновских вязких жидкостей. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 217–234. DOI: 10.15514/ISPRAS–2025–37(1)–13.

The LS-STAG Immersed Boundary Method Modification for Non-Newtonian Viscous Fluids Computation

[1] *I. K. Marchevsky*, ORCID: 0000-0003-4899-4828 <*iliamarchevsky@mail.ru*>
[2] *V.V. Puzikova*, ORCID: 0000-0003-0712-4519 <*v.puzikova@yadro.com*>

[1] *Bauman Moscow State Technical University,*
5, 2nd Baumanskaya st., Moscow, 105005, Russia.
[2] *YADRO, 15 bld.15, Rochdelskaya st., Moscow, 123022, Russia.*

Abstract. A modification of the immersed boundary method LS-STAG with level functions has been developed to simulate flows of non-Newtonian viscous fluids. The viscosity of these fluids is completely determined by the intensity of the strain rate tensor at each point at any time. The LS-STAG method modification was previously developed for another class of non-Newtonian fluids (viscoelastic fluids). The modification demonstrated high accuracy even for viscoelactic flows characterized by high values of the Weissenberg number. Therefore, it is of interest to generalize the LS-STAG method for non-Newtonian viscous fluids. Note that although the LS-STAG method can be successfully used to simulate flows with moving immersed boundaries, this paper focuses only on flows with fixed ones. The developed numerical method can be used both for viscoplastic fluids and for generalized Newtonian fluids that do not have a yield point. For viscoplastic fluids, the Ofoli-Morgan-Steffe, Mizrahi-Burk, Casson, and Herschel-Bulkley models used with the Bercovier-Engelmann and Papanastasiou regularization models are considered, and for fluids that do not have a yield point, the Ellis, Cross, Carreau, Yasuda (Carreau-Yasuda) models, as well as the Ostwald-de Waele power model are considered. To verify the numerical method developed and implemented in the author's software package, a well-studied problem of power-law flow past a stationary circular airfoil was used at different values of the Reynolds number and flow index. The obtained results are in good agreement with the known in the literature computational data of other researchers. In the future, it is planned to generalize the developed modification of the LS-STAG method for non-Newtonian viscous fluids simulation for the case of moving immersed boundaries.

Keywords: immersed boundary method; the LS-STAG method; non-Newtonian viscous fluid; power-law fluid; airfoil; level-set function.

For citation: Marchevsky I.K., Puzikova V.V. The LS-STAG Immersed Boundary Method Modification for Non-Newtonian Viscous Fluids Computation. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 217-234 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-13.

1. Введение

Неньютоновские вязкие жидкости или обобщенные ньютоновские жидкости – жидкости, для которых вязкость ν в каждой точке в любой момент времени полностью определяется интенсивностью $\dot{\gamma}$ тензора скоростей деформации $\hat{S}$ в той же точке в тот же самый момент времени, т.е. вязкость ν является функцией одной переменной $(\dot{\gamma})$ и явно не зависит от координат и времени действия напряжений (предыстории жидкости). Общим свойством таких жидкостей является близость к реологическому поведению ньютоновских жидкостей при малых значениях $\dot{\gamma}$: при этих значениях на графике функции $\nu(\dot{\gamma})$ наблюдается так называемое ньютоновское плато. Обобщенные ньютоновские жидкости делятся на две группы [1, 2] в зависимости от наличия предела текучести или предельного напряжения сдвига – напряжения, при котором начинает развиваться необратимая пластическая деформация. Это неньютоновские вязкие жидкости, не обладающие пределом текучести, например, нефть, вулканическая лава, грязевые сели, и вязкопластичные жидкости, такие как керамическая паста, шпатлевка, известка.

Поскольку вязкопластичные жидкости обладают пределом текучести $\tau_0 \neq 0$, они ведут себя как твердые материалы, а при воздействии проявляют поведение жидкостей. С помощью предела текучести для вязкопластичных сред рассчитываются допустимые напряжения: после прохождения предела текучести в среде начинают происходить необратимые

изменения, сильно изменяется взаимное расположение молекул или частиц среды, появляются значительные пластические деформации. Если среда является металлом, то перестраивается кристаллическая решетка и металл самоупрочняется: после прохождения предела текучести деформации растут при возрастающем значении растягивающей силы.

Отсутствие предела текучести τ_0 означает, что необратимые деформации наблюдаются при любой нагрузке. Таким образом, для неньютоновских вязких жидкостей, не обладающих пределом текучести, график зависимости напряжений от скоростей деформации проходит через начало координат, т.е. скорости деформации равны нулю, когда напряжения равны нулю. По характеру монотонности функции $\nu(\dot{\gamma})$ такие жидкости делятся на две группы: псевдопластичные и дилатантные. В первом случае $\nu(\dot{\gamma})$ является убывающей функцией (вязкость снижается при увеличении нагрузки, т.е. происходит разжижение), а во втором – возрастающей (вязкость возрастает при увеличении нагрузки, т.е. происходит загустевание). Для неньютоновских жидкостей возможны только эти два случая, поскольку если при отсутствии предельного напряжения τ_0 вязкость жидкости одновременно не возрастает и не убывает при всех значениях $\dot{\gamma}$, т.е. является константой, то жидкость является ньютоновской.

Примерами псевдопластичных жидкостей являются целлюлозные лаки, смазки, нефть, вулканическая лава, грязевые сели, высококонцентрированные растворы полимеров, резины, крахмала, вискоза, латекс, эмульсии, покрытия. К дилатантным жидкостям относятся растворы жидкой глины, песка или бетона, высококонцентрированные суспензии, клеи, покрытия смеси извести с водой, мед, некоторые компоненты леденцов. При малых скоростях жидкость действует как смазка, и дилатантная жидкость течет легко. При более высоких скоростях жидкость уже не успевает заполнить пространство между частицами, и трение значительно увеличивается, вызывая увеличение вязкости. Благодаря описанным свойствам дилатантные жидкости применяют при изготовлении бронежилетов. Такая система обеспечивает владельцу достаточную гибкость для нормального диапазона движения, а также жесткость при попадании пули, колющих ударов ножом, и т.п. Принцип действия такого жилета похож на эффект кольчуги, но бронежилет с дилатантными жидкостями намного легче. Такая жидкость распределяет силу быстрого внезапного удара по более широкой области тела пользователя, снижая травматичность удара. Однако, против медленных, но сильных ударов, при которых дилатантная жидкость имеет низкую вязкость и может течь, такие жидкости дополнительной защиты не обеспечивают.

Целью данной работы является разработка модификации метода погруженных границ [3] с усеченными ячейками и функциями уровня [4] LS-STAG [5] для моделирования течений неньютоновских вязких жидкостей, ее программная реализация и верификация. Характерной особенностью методов погруженных границ является то, что сетка в них не связана с обтекаемым телом, что делает их особенно удобными при моделировании течений в областях со сложной геометрией. Кроме того, данная особенность позволяет производить расчет течений с подвижными границами без перестроения сетки на каждом шаге расчета. Благодаря использованию аппарата функций уровня метод LS-STAG в отличие от других методов погруженных границ позволяет строить дискретные аналоги уравнений единообразно для всех типов ячеек, как полностью заполненных жидкостью или телом, так и усеченных, которые содержат одновременно и жидкость, и твердое тело. Это позволяет не разделять жидкие и твердые ячейки при составлении систем линейных алгебраических уравнений. При этом в двумерном случае шаблон дискретизации имеет пятиточечную структуру. Кроме того, использование функций уровня позволяет легко вычислять все необходимые геометрические характеристики ячеек сетки, в результате чего затраты машинного времени на обработку ячеек сложной формы уменьшаются. В основе построения дискретных аналогов уравнений в методе LS-STAG лежат численные аналоги законов сохранения массы, импульса и кинетической энергии, а на усеченных ячейках корректно учитываются граничные условия, что позволяет получать физически правдоподобное численное решение.

Ранее была разработана модификация метода LS-STAG для другого класса неньютоновских жидкостей – вязкоупругих жидкостей, показавшая высокую точность даже для течений, которые характеризуются высокими значениями числа Вайсенберга [6-8]. Поэтому представляет интерес обобщить метод LS-STAG и для описанных выше неньютоновских вязких жидкостей. Отметим, что несмотря на то, что метод LS-STAG может успешно применяться для моделирования течений с подвижными погруженными границами [9-11], в данной работе внимание сфокусировано только на течениях с неподвижными границами. Для верификации метода используется хорошо исследованная задача об обтекании неподвижного кругового профиля.

2. Постановка тестовой задачи

Рассмотрим внешнее обтекание неподвижного кругового профиля с границей K и диаметром D (рис. 1) равномерным потоком неньютоновской вязкой жидкости постоянной плотности в расчетной области $\Omega = [0, 23D] \times [0, 24D]$ с внешней границей $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4$.

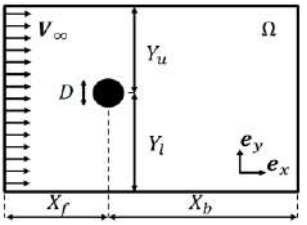


Рис. 1. Расчетная область.
Fig. 1. Computational domain.

Пусть $D = 1$, а центр профиля находится на расстоянии $X_f = 8$ от левой границы расчетной области, на $X_b = 15$ – от правой, и на расстоянии $Y_u = Y_l = 12$ от верхней и нижней границы. В безразмерных переменных математическая постановка задачи имеет вид:

$$\begin{cases} \nabla \cdot \vec{v} = 0, \quad \dfrac{\partial \vec{v}}{\partial t} + (\vec{v} \cdot \nabla)\vec{v} + \nabla p = \nabla \cdot \hat{\tau}, \\ \vec{v}(x, y, 0) = \vec{v}_0(x, y), \quad (x, y) \in \Omega, \\ \vec{v}\big|_{\Gamma_1} = \vec{v}\big|_{\Gamma_2} = \vec{v}\big|_{\Gamma_3} = \vec{V}_\infty, \quad \vec{v}\big|_K = 0, \quad \left.\dfrac{\partial \vec{v}}{\partial \vec{n}}\right|_{\Gamma_4} = \vec{0}, \quad \left.\dfrac{\partial p}{\partial \vec{n}}\right|_{\Gamma \cup K} = 0. \end{cases}$$

Здесь t – безразмерное время; x, y – безразмерные координаты; p – безразмерное давление; n – внешняя нормаль; $\vec{v} = u \cdot \vec{e}_x + v \cdot \vec{e}_y$ – безразмерная скорость; $\hat{\tau}$ – тензор напряжений

$$\hat{\tau} = 2\nu(\dot{\gamma})\hat{S};$$

$\hat{S}$ – тензор скоростей деформации

$$\hat{S} = \frac{1}{2}(\nabla\vec{v} + [\nabla\vec{v}]^T) = \begin{pmatrix} \dfrac{\partial u}{\partial x} & \dfrac{1}{2}\left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x}\right) \\ \dfrac{1}{2}\left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x}\right) & \dfrac{\partial v}{\partial y} \end{pmatrix};$$

$\dot{\gamma}$ – интенсивность (второй инвариант) тензора скоростей деформации

$$\dot{\gamma} = \sqrt{\frac{1}{2}\hat{S}:\hat{S}} = \sqrt{2 \cdot \left\{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right\} + \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2};$$

$\nu(\dot{\gamma})$ – вязкость неньютоновской жидкости, зависящая от интенсивности тензора скоростей деформации. Вид этой зависимости определяется моделью неньютоновской жидкости.

2.1. Модели вязких неньютоновских жидкостей

Для вязкопластичных жидкостей с пределом текучести τ_0 рассмотрим наиболее общую модель – модель Офоли-Моргана-Штеффе [12-14] с регуляризацией:

$$\nu(\dot{\gamma}) = \left(g_o(\dot{\gamma}, \varepsilon, m) \cdot \sqrt[m]{\frac{\tau_0}{\dot{\gamma}}} + \sqrt[m]{\nu_0 \dot{\gamma}^{n-1}} \right)^m .$$

Здесь $m > 0$ – параметр, который определяет плавность перехода от зоны твердого тела к зоне вязкопластичного течения: чем больше значение m, тем более плавным становится переход; n – индекс течения, как и в случае неньютоновских вязких жидкостей, не обладающих пределом текучести; ε – параметр регуляризации; функция $g_o(\dot{\gamma}, \varepsilon, m)$ определяется выбранной моделью регуляризации (табл. 1).

Табл. 1. Модели регуляризации для вязкопластичных жидкостей.
Table 1. Regularization models for viscoplastic fluids.

Модель	$g_o(\dot{\gamma}, \varepsilon, m)$	$g(\dot{\gamma}, \varepsilon)$
Берковьера-Энгельмана [15]	$\sqrt[m]{g(\dot{\gamma}, \varepsilon)}$	$\frac{\dot{\gamma}}{\dot{\gamma}+\varepsilon}$ или $\frac{\dot{\gamma}}{\sqrt{\dot{\gamma}^2+\varepsilon^2}}$
Папанастасио [16]	$g\left(\sqrt[m]{\dot{\gamma}}, \sqrt[m]{\varepsilon}\right)$	$1 - \exp\left(-\frac{\dot{\gamma}}{\varepsilon}\right)$

Использование модели регуляризации позволяет вычислять компоненты тензора напряжений по единой формуле как для областей вязкопластичного течения, так и для областей твердого тела. При использовании всех моделей регуляризации (табл. 1) имеем:

$$\nu \to \nu_0^e = \left(\sqrt[m]{\frac{\tau_0}{\varepsilon}} + \sqrt[m]{\nu_0} \right)^m \text{ при } \dot{\gamma} \to 0.$$

В зависимости от индекса течения модель Офоли-Моргана-Штеффе с регуляризацией может описывать следующие вязкопластичные жидкости:

- разжижающиеся вязкопластичные жидкости (при $n < 1$);
- бингамовские вязкопластичные жидкости (при $n = 1$);
- загустевающие вязкопластичные жидкости (при $n > 1$).

Отметим, что при $m = 2$ модель Офоли-Моргана-Штеффе принимает вид модели Мизрахи-Берка [14, 17], если же одновременно имеем $m = 2$ и $n = 1$ – модели Кассо [13]. При $m = 1$ рассматриваемая модель принимает вид модели Гершеля-Балкли [18], а если одновременно имеем $m = 1$ и $n = 1$ – модели Бингама [19].

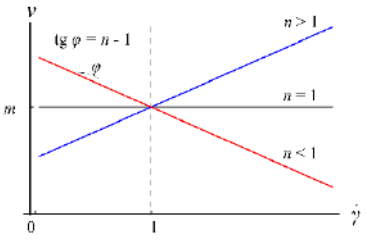
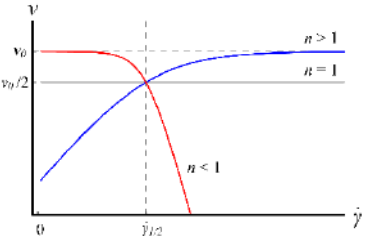
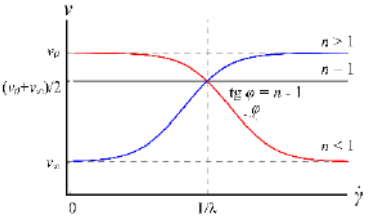
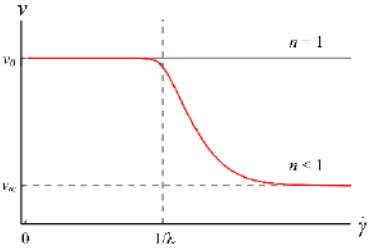
Для неньютоновских вязких жидкостей, не обладающих пределом текучести, зависимости $\nu(\dot{\gamma})$ для различных моделей представлены в табл. 2.

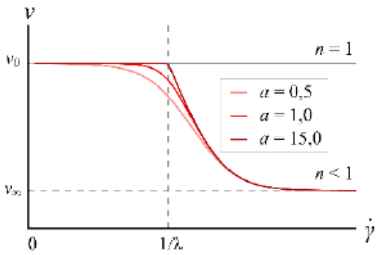
3. Основные идеи модифициированного метода LS-STAG

3.1 Модификация LS-STAG-сетки

В расчетной области Ω вводится прямоугольная сетка с ячейками $\Omega_{i,j} = (x_{i-1}, x_i) \times (y_{j-1}, y_j)$, площади которых равны $V_{i,j} = \Delta x_i \Delta y_j$. Радиусы-векторы центров ячеек сетки будем обозначать $\vec{x}_{i,j}^c = (x_i^c, y_j^c)$, а границы – $\Gamma_{i,j}$ (рис. 2). Ячейка $\Omega_{i,j}$ данной сетки, которая далее будет называться «основной», является контрольным объёмом, который используется для дискретизации уравнения неразрывности и уравнений для нормальных неньютоновских вязкоупругих напряжений.

Табл. 2. Модели неньютоновских вязких жидкостей, не обладающих пределом текучести.
Table 2. Models of non-Newtonian viscous fluids that do not have a yield stress.

Модель	$\nu(\dot{\gamma})$	Псевдопластичные	Дилатантные
Степенная [20] (Оствальда – де Веле)	$\nu = m\dot{\gamma}^{n-1}$  m – мера консистентности (показатель консистентности или консистентность) жидкости; n – индекс течения.	$n < 1$: модель адекватна для вулканической лавы, грязевых селей (n от 0,1 до 0,4), нефти ($n = 0{,}8$).	$n > 1$: модель адекватна для растворов и сплавов полимеров, суспензий, красок, меда ($n = 2{,}5$), сыпучих материалов, смеси извести с водой ($n = 1{,}47$).
Эллиса [21]	$\nu = \dfrac{\nu_0}{1 + \left(\dfrac{\dot{\gamma}}{\dot{\gamma}_{1/2}}\right)^{\alpha-1}}$  $\dot{\gamma} \to \dot{\gamma}_{1/2}$ при $\nu \to \nu_0/2$; $\alpha = 1/n$.	$n < 1$; $\nu \to \nu_0$ при $\dot{\gamma} \to 0$.	$n > 1$; $\nu \to \nu_0$ при $\dot{\gamma} \to \infty$.
Кросса [22]	$\nu = \nu_\infty + \dfrac{\nu_0 - \nu_\infty}{1 + (\lambda\dot{\gamma})^{1-n}}$  λ – время релаксации.	$n < 1$; $\nu \to \nu_0$ при $\dot{\gamma} \to 0$; $\nu \to \nu_\infty$ при $\dot{\gamma} \to \infty$.	$n > 1$; $\nu \to \nu_\infty$ при $\dot{\gamma} \to 0$; $\nu \to \nu_0$ при $\dot{\gamma} \to \infty$.
Карро [23]	$\nu = \nu_\infty + \dfrac{\nu_0 - \nu_\infty}{[1 + (\lambda\dot{\gamma})^2]^{(1-n)/2}}$  λ – время релаксации.	$n < 1$; $\nu \to \nu_0$ при $\dot{\gamma} \to 0$; $\nu \to \nu_\infty$ при $\dot{\gamma} \to \infty$.	–

Модель	$\nu(\dot{\gamma})$	Псевдопластичные	Дилатантные
Язуды [24] (Карро-Язуды)	$\nu = \nu_\infty + \frac{\nu_0 - \nu_\infty}{[1 + (\lambda\dot{\gamma})^a]^{(1-n)/a}}$  a – параметр, характеризующий ширину переходной области между плато на уровне ν_0 и степенной областью.	$n < 1$; $\nu \to \nu_0$ при $\dot{\gamma} \to 0$; $\nu \to \nu_\infty$ при $\dot{\gamma} \to \infty$; при $a = 2$ модель Язуды совпадает с моделью Карро.	–

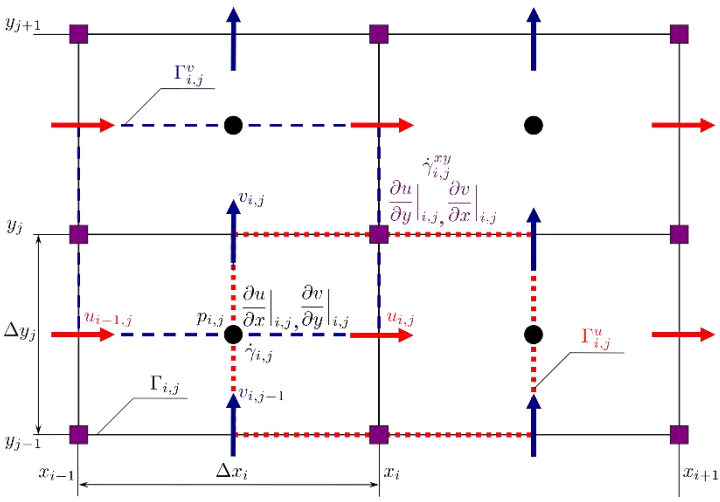


Рис. 2. Разнесенные сетки.
Fig. 2. Staggered meshes.

Вместе с «основной» строятся смещенные сетки с ячейками $\Omega^u_{i,j} = (x^c_i, x^c_{i+1}) \times \left(y_{j-1}, y_j\right)$ и $\Omega^v_{i,j} = (x_{i-1}, x_i) \times \left(y^c_j, y^c_{j+1}\right)$, границы которых обозначим $\Gamma^u_{i,j}$ и $\Gamma^v_{i,j}$ соответственно (рис. 2). Эти ячейки являются контрольными объемами для уравнения баланса импульса в проекциях на оси Ox и Oy. Если $i = \overline{1,N}$, $j = \overline{1,M}$, то основная сетка содержит $G = N \cdot M$ ячеек, x-сетка – $G_x = (N-1) \cdot M$ ячеек, а y-сетка – $G_y = N \cdot (M-1)$.

Для описания положения границы Γ^{ib} твердого тела произвольной формы Ω^{ib} вводят знакопеременную функцию расстояния $\varphi(\vec{r})$ (функцию уровня), такую что

$$\begin{cases} \varphi(\vec{r}) < 0, \ \vec{r} \in \Omega^f \\ \varphi(\vec{r}) = 0, \ \vec{r} \in \Gamma^{ib} \\ \varphi(\vec{r}) > 0, \ \vec{r} \in \Omega^{ib} \end{cases}$$

В рассматриваемом случае обтекания кругового профиля с центром в точке (x_C, y_C) функция уровня может быть задана аналитически:

$$\varphi(x,y) = R - \sqrt{(x - x_C)^2 + (y - y_C)^2}.$$

В каждой усеченной ячейке $\Omega_{i,j}$ LS-STAG-сетки погруженная граница $\Gamma_{i,j}^{ib}$ представляется отрезком прямой, положения концов которого определяются линейной интерполяцией величины $\varphi_{i,j}$, принимающей значение функции уровня $\varphi(x_i, y_j)$ в правом верхнем углу ячейки $\Omega_{i,j}$. Для определения типа усеченной ячейки вводят коэффициенты заполнения ячеек $\vartheta_{i,j}^u$, $\vartheta_{i,j}^v \in [0,1]$, показывающие, какая часть ячейки занята жидкостью на восточной и северной гранях ячейки соответственно.

В двумерном случае все усеченные ячейки можно разделить на три группы: трапециевидные, треугольные и пятиугольные. Примеры всех типов ячеек представлены на рис. 3.

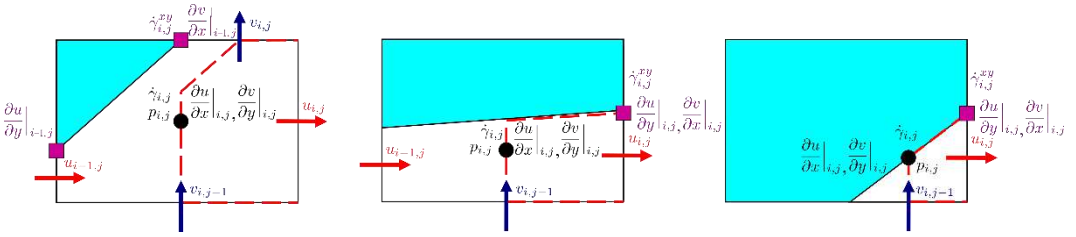


Рис. 3. Положение точек вычисления давления, скоростей и напряжений для основных типов усеченных ячеек LS-STAG-сетки: северо-западной треугольной ячейки (слева), северной трапециевидной ячейки (в центре) и северо-западной пятиугольной ячейки (справа).

Fig. 3. Positions of pressure, velocity and stress computation points for the main types of LS-STAG cut-cells: northwest triangular cell (left), north trapezoidal cell (center) and northwest pentagonal cell (right).

Положения точек, в которых вычисляются неизвестные величины, зависят от типа ячейки $\Omega_{i,j}$ (рис. 2 и 3). Значения скоростей $u_{i,j}$ и $v_{i,j}$ вычисляются в серединах жидких частей граней, а давление и нормальные напряжения аппроксимируется кусочно-постоянной функцией на каждой ячейке (для удобства на чертежах привязываем точку вычисления значений $p_{i,j}$, $\left.\frac{\partial u}{\partial x}\right|_{i,j}$ и $\left.\frac{\partial v}{\partial y}\right|_{i,j}$ к центру ячейки основной сетки). Точки вычисления касательных напряжений $\left.\frac{\partial u}{\partial y}\right|_{i,j}$ и $\left.\frac{\partial v}{\partial x}\right|_{i,j}$ зависят от типа ячейки.

Интенсивность тензора скоростей деформации вычисляется как в точках вычислениях нормальных напряжений (обозначается $\dot{\gamma}_{i,j}$), так и в точках вычислениях касательных напряжений (обозначается $\dot{\gamma}_{i,j}^{xy}$). Каждой ячейке $\Omega_{i,j}$ основной сетки присвоим вес $\alpha_{i,j}$:

$$\alpha_{i,j} = \begin{cases} 0, \text{если } \Omega_{i,j} - \text{твердая ячейка}, \\ 1/3, \text{если } \Omega_{i,j} - \text{треугольная ячейка}, \\ 1/4, \quad \text{в остальных случаях}. \end{cases}$$

Тогда

$$\dot{\gamma}_{i,j} = \sqrt{2\left\{\left(\left.\frac{\partial u}{\partial x}\right|_{i,j}\right)^2 + \left(\left.\frac{\partial v}{\partial y}\right|_{i,j}\right)^2\right\} + \alpha_{i,j}\left(\tau_{i,j}^{s,sqr} + \tau_{i-1,j}^{s,sqr} + \tau_{i,j-1}^{s,sqr} + \tau_{i-1,j-1}^{s,sqr}\right)};$$

$$\dot{\gamma}_{i,j}^{xy} = \sqrt{2\left\{\left(\left.\overline{\frac{\partial u}{\partial x}}\right|_{i,j}\right)^2 + \left(\left.\overline{\frac{\partial v}{\partial y}}\right|_{i,j}\right)^2\right\} + \tau_{i,j}^{s,sqr}}.$$

Здесь $\tau_{i,j}^{s,sqr} = \left(\left.\frac{\partial u}{\partial y}\right|_{i,j} + \left.\frac{\partial v}{\partial x}\right|_{i,j}\right)^2$; для величины z, вычисляемой в центрах ячеек «основной» сетки, значение $\overline{z}_{i,j}$ определяется следующим образом:

$$\overline{z}_{i,j} = \frac{\alpha_{i,j}V_{i,j}z_{i,j} + \alpha_{i+1,j}V_{i+1,j}z_{i+1,j} + \alpha_{i,j+1}V_{i,j+1}z_{i,j+1} + \alpha_{i+1,j+1}V_{i+1,j+1}z_{i+1,j+1}}{\alpha_{i,j}V_{i,j} + \alpha_{i+1,j}V_{i+1,j} + \alpha_{i,j+1}V_{i,j+1} + \alpha_{i+1,j+1}V_{i+1,j+1}}.$$

Для вычисления компонент тензора скоростей деформации необходимо построить дискретные аналоги нормальных и касательных напряжений. Значения нормальных напряжений $\left.\frac{\partial u}{\partial x}\right|_{i,j}$ и $\left.\frac{\partial v}{\partial y}\right|_{i,j}$ независимо от типа ячейки $\Omega_{i,j}$ вычисляются по формулам

$$\left.\frac{\partial u}{\partial x}\right|_{i,j} \approx \frac{\vartheta^u_{i,j}u_{i,j} - \vartheta^u_{i-1,j}u_{i-1,j} + \left(\vartheta^u_{i-1,j} - \vartheta^u_{i,j}\right)u^{ib}_{i,j}}{V_{i,j}/\Delta y_j},$$
$$\left.\frac{\partial v}{\partial y}\right|_{i,j} \approx \frac{\vartheta^v_{i,j}v_{i,j} - \vartheta^v_{i,j-1}v_{i,j-1} + \left(\vartheta^v_{i,j-1} - \vartheta^v_{i,j}\right)v^{ib}_{i,j}}{V_{i,j}/\Delta x_i}.$$

Эти формулы верны для усеченных ячеек любого типа, в них естественным образом учитываются граничные условия. Для жидких ячеек они переходят в стандартные конечно-разностные соотношения. Формулы для расчета значений касательных напряжений $\left.\frac{\partial u}{\partial y}\right|_{i,j}$ и $\left.\frac{\partial v}{\partial x}\right|_{i,j}$ зависят от типов ячеек: если одновременно $\vartheta^u_{i,j} \neq 0$ и $\vartheta^u_{i,j+1} \neq 0$, то $\left.\frac{\partial u}{\partial y}\right|_{i,j}$ вычисляется по формуле

$$\left.\frac{\partial u}{\partial y}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y^{xy}_{i,j}},$$

где

$$\Delta y^{xy}_{i,j} = \frac{1}{2}\vartheta^u_{i,j+1}\Delta y_{j+1} + \frac{1}{2}\vartheta^u_{i,j}\Delta y_j,$$

иначе

$$\vartheta^u_{i,j} = 0 \quad \Rightarrow \quad \left.\frac{\partial u}{\partial y}\right|_{i,j} = \left.\frac{\partial u}{\partial y}\right|^{ib,s}_{i,j} = \frac{u_{i,j+1} - u\left(x_i, y^{ib}_{i,j+1}\right)}{\frac{1}{2}\vartheta^u_{i,j+1}\Delta y_{j+1}},$$

$$\vartheta^u_{i,j+1} = 0 \quad \Rightarrow \quad \left.\frac{\partial u}{\partial y}\right|_{i,j} = \left.\frac{\partial u}{\partial y}\right|^{ib,n}_{i,j} = \frac{u\left(x_i, y^{ib}_{i,j}\right) - u_{i,j}}{\frac{1}{2}\vartheta^u_{i,j}\Delta y_j};$$

если одновременно $\vartheta^v_{i,j} \neq 0$ и $\vartheta^v_{i+1,j} \neq 0$, то $\left.\frac{\partial v}{\partial x}\right|_{i,j}$ вычисляется по формуле

$$\left.\frac{\partial v}{\partial x}\right|_{i,j} = \frac{v_{i+1,j} - v_{i,j}}{\Delta x^{xy}_{i,j}},$$

где

$$\Delta x^{xy}_{i,j} = \frac{1}{2}\vartheta^v_{i+1,j}\Delta x_{i+1} + \frac{1}{2}\vartheta^v_{i,j}\Delta x_i,$$

иначе

$$\vartheta^v_{i,j} = 0 \quad \Rightarrow \quad \left.\frac{\partial v}{\partial x}\right|_{i,j} = \left.\frac{\partial v}{\partial x}\right|^{ib,w}_{i,j} = \frac{v_{i+1,j} - v\left(x^{ib}_{i+1,j}, y_j\right)}{\frac{1}{2}\vartheta^v_{i+1,j}\Delta x_{i+1}},$$

$$\vartheta^v_{i+1,j} = 0 \quad \Rightarrow \quad \left.\frac{\partial v}{\partial x}\right|_{i,j} = \left.\frac{\partial v}{\partial x}\right|^{ib,e}_{i,j} = \frac{v\left(x^{ib}_{i,j}, y_j\right) - v_{i,j}}{\frac{1}{2}\vartheta^v_{i,j}\Delta x_i}.$$

3.2. Особенности построения дискретных аналогов уравнений неразрывности и баланса импульса

В данной работе рассматривается только случай неподвижных погруженных границ. Пусть Ω^* – некоторая область, Γ^* – ее граница. Тогда уравнение неразрывности и уравнение баланса импульса в проекциях на оси Ox и Oy соответственно с учетом формул для тензоров напряжений и скоростей деформации можно переписать в интегральной форме:

$$\int_{\Gamma^*} \vec{v} \cdot \vec{n}\, dS = 0,$$

$$\frac{d}{dt}\int_{\Omega^*} u\, dV + \int_{\Gamma^*} (\vec{v}\cdot\vec{n})\, u\, dS + \int_{\Gamma^*} p\,\overrightarrow{e_x}\cdot\vec{n}\, dS - \boxed{\int_{\Gamma^*} \nu(\dot{\gamma})\left[\frac{\partial u}{\partial x}\vec{e}_x + \frac{\partial u}{\partial y}\vec{e}_y\right]\cdot\vec{n}\, dS} - \boxed{\int_{\Gamma^*} \nu(\dot{\gamma})\left[\frac{\partial u}{\partial x}\vec{e}_x + \frac{\partial v}{\partial x}\vec{e}_y\right]\cdot\vec{n}\, dS} = 0,$$

$$\frac{d}{dt}\int_{\Omega^*} v\, dV + \int_{\Gamma^*} (\vec{v}\cdot\vec{n})\, v\, dS + \int_{\Gamma^*} p\,\overrightarrow{e_y}\cdot\vec{n}\, dS - \boxed{\int_{\Gamma^*} \nu(\dot{\gamma})\left[\frac{\partial v}{\partial x}\vec{e}_x + \frac{\partial v}{\partial y}\vec{e}_y\right]\cdot\vec{n}\, dS} - \boxed{\int_{\Gamma^*} \nu(\dot{\gamma})\left[\frac{\partial u}{\partial y}\vec{e}_x + \frac{\partial v}{\partial y}\vec{e}_y\right]\cdot\vec{n}\, dS} = 0.$$

Отличие от аналогичных уравнений для ньютоновской жидкости заключается в обведенных рамками членах, описывающих вязкую диффузию. При построении разностных аналогов для получения независимых систем линейных алгебраических уравнений для u и v необходимо сделать различие между членами, обведенными в уравнениях выше одинарными и двойными рамками: значения всех переменных, которые входят в слагаемые, обведенные одинарными рамками, необходимо брать с предыдущего шага по времени, а в слагаемые, обведенные двойными рамками, будут входить неизвестные значения u и v. Тогда для уравнений баланса импульса дискретизированная по пространству разностная схема будет иметь следующий матричный вид:

$$\frac{\mathrm{d}}{\mathrm{dt}}(\mathrm{M^x u}) + \mathrm{C^x}[\overline{\mathrm{u}}]\mathrm{u} + \mathrm{G^x P} - \mathrm{K^{\tau,x} u} + \mathrm{S_x^{ib,c}} - \mathrm{S_{\tau,x}^{ib,\nu}} - \mathrm{F_u^d} = 0,$$

$$\frac{d}{dt}(M^y v) + C^y[\overline{v}]v + G^y P - K^{\tau,y} v + S_y^{ib,c} - S_{\tau,y}^{ib,\nu} - F_v^d = 0.$$

Здесь $P \in \mathbb{R}^G$ – дискретный аналог давления; $u \in \mathbb{R}^{G_x}$ – дискретный аналог проекции скорости на ось Ox; $v \in \mathbb{R}^{G_y}$ – дискретный аналог проекции скорости на ось Oy; элементы диагональных матриц $M^x \in M(\mathbb{R})_{G_x\times G_x}$ и $M^y \in M(\mathbb{R})_{G_y\times G_y}$ являются объемами ячеек $\Omega_{i,j}^u$ и $\Omega_{i,j}^v$ соответственно; $C^x[\overline{u}] \in M(\mathbb{R})_{G_x\times G_x}$ и $C^y[\overline{v}] \in M(\mathbb{R})_{G_y\times G_y}$ задают дискретные аналоги конвективных потоков, $G^x \in M(\mathbb{R})_{G_x\times G}$ и $G^y \in M(\mathbb{R})_{G_y\times G}$ – градиента давления; матрицы $K^{\tau,x} \in M(\mathbb{R})_{G_x\times G_x}$ и $K^{\tau,y} \in M(\mathbb{R})_{G_y\times G_y}$ возникают из дискретных аналогов членов, отвечающих за вязкую диффузию и обведенных в уравнениях выше двойными рамками; $S_x^{ib,c} \in \mathbb{R}^{G_x}$ и $S_y^{ib,c} \in \mathbb{R}^{G_y}$ – источниковые члены, возникающие в силу граничных условий из конвективных членов; $S_{\tau,x}^{ib,\nu} \in \mathbb{R}^{G_x}$ и $S_{\tau,y}^{ib,\nu} \in \mathbb{R}^{G_y}$ – источниковые члены, возникающие в силу граничных условий из вязких членов, обведенных в уравнениях выше двойными рамками; $F_u^d \in \mathbb{R}^{G_x}$ и $F_v^d \in \mathbb{R}^{G_y}$ – дискретные аналоги вязких членов, обведенных в уравнениях выше одинарными рамками. Дискретный аналог уравнения неразрывности в матричной форме, как и в случае ньютоновской жидкости, имеет вид

$$D^x u + D^y v + \overline{U}^{ib} = 0.$$

Здесь $D^x = -(G^x)^T \in M(\mathbb{R})_{G\times G_x}$; $D^y = -(G^y)^T \in M(\mathbb{R})_{G\times G_y}$; $\overline{U}^{ib} \in \mathbb{R}^G$ – источниковый член, возникающий в силу граничных условий.

Формулы для вычисления элементов матриц M^x, M^y, $C^x[\overline{u}]$, $C^y[\overline{v}]$, D^x, D^y и векторов $S_x^{ib,c}$, $S_y^{ib,c}$, $\overline{U}^{ib}$ совпадают с аналогичными для случая ньютоновской жидкости и могут быть взяты из [5]. Таким образом, для случая неньютоновских вязких жидкостей необходимо вывести только формулы для вычисления элементов матриц $K^{\tau,x}$, $K^{\tau,y}$ и векторов $S_{\tau,x}^{ib,\nu}$, $S_{\tau,y}^{ib,\nu}$, F_u^d, F_v^d.

С учетом представленных выше формул для напряжений, элементы векторов F_u^d и F_v^d можно вычислять по формулам, единым для всех типов ячеек LS-STAG-сетки:

$$\begin{aligned}
F_u^d|_{i,j} &= \left[\nu\left(\dot\gamma_{i+1,j}\right)\left.\frac{\partial u}{\partial x}\right|_{i+1,j} - \nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial u}{\partial x}\right|_{i,j}\right]\vartheta_{i,j}^u \Delta y_j + \\
&+\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta x_{i,j}^{xy}\left.\frac{\partial v}{\partial x}\right|_{i,j} - \nu\left(\dot\gamma_{i,j-1}^{xy}\right)\Delta x_{i,j-1}^{xy}\left.\frac{\partial v}{\partial x}\right|_{i,j-1}, \\
F_v^d|_{i,j} &= \left[\nu\left(\dot\gamma_{i,j+1}\right)\left.\frac{\partial v}{\partial y}\right|_{i,j+1} - \nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial v}{\partial y}\right|_{i,j}\right]\vartheta_{i,j}^v \Delta x_i + \\
&+\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta y_{i,j}^{xy}\left.\frac{\partial u}{\partial y}\right|_{i,j} - \nu\left(\dot\gamma_{i-1,j}^{xy}\right)\Delta y_{i-1,j}^{xy}\left.\frac{\partial u}{\partial y}\right|_{i-1,j}.
\end{aligned}$$

Аналогично получаем дискретные аналоги членов из уравнения баланса импульса, обведенных двойными рамками:

$$\begin{aligned}
\int\limits_{\Gamma_{i,j}^u} \nu\left(\dot\gamma\right)\left[\frac{\partial u}{\partial x}\vec{e}_x + \frac{\partial u}{\partial y}\vec{e}_y\right]\cdot\vec{n}\,dS &\approx \left[\nu\left(\dot\gamma_{i+1,j}\right)\left.\frac{\partial u}{\partial x}\right|_{i+1,j} - \nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial u}{\partial x}\right|_{i,j}\right]\vartheta_{i,j}^u \Delta y_j + \\
&+\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta x_{i,j}^{xy}\left.\frac{\partial u}{\partial y}\right|_{i,j} - \nu\left(\dot\gamma_{i,j-1}^{xy}\right)\Delta x_{i,j-1}^{xy}\left.\frac{\partial u}{\partial y}\right|_{i,j-1}, \\
\int\limits_{\Gamma_{i,j}^v} \nu\left(\dot\gamma\right)\left[\frac{\partial v}{\partial x}\vec{e}_x + \frac{\partial v}{\partial y}\vec{e}_y\right]\cdot\vec{n}\,dS &\approx \left[\nu\left(\dot\gamma_{i,j+1}\right)\left.\frac{\partial v}{\partial y}\right|_{i,j+1} - \nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial v}{\partial y}\right|_{i,j}\right]\vartheta_{i,j}^v \Delta x_i + \\
&+\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta y_{i,j}^{xy}\left.\frac{\partial v}{\partial x}\right|_{i,j} - \nu\left(\dot\gamma_{i-1,j}^{xy}\right)\Delta y_{i-1,j}^{xy}\left.\frac{\partial v}{\partial x}\right|_{i-1,j}.
\end{aligned}$$

Поставим в формулы для элементов векторов F_u^d и F_v^d формулы выражения для напряжений. Тогда получим, что дискретизация членов из уравнения баланса импульса, обведенных двойными рамками, записывается на пятиточечном шаблоне следующим образом:

$$\begin{aligned}
\int\limits_{\Gamma_{i,j}^u} \nu(\dot\gamma)\left[\frac{\partial u}{\partial x}\overrightarrow{e_x} + \frac{\partial u}{\partial y}\overrightarrow{e_y}\right]\cdot\vec{n}\,dS &\approx K_W^{\tau,x}(i,j)u_{i-1,j} + K_E^{\tau,x}(i,j)u_{i+1,j} + K_P^{\tau,x}(i,j)u_{i,j} + \\
&+K_S^{\tau,x}(i,j)u_{i,j-1} + K_N^{\tau,x}(i,j)u_{i,j+1} + S_{\tau,x|_{i,j}}^{ib,\nu},
\end{aligned}$$

$$K_P^{\tau,x}(i,j) = -\frac{\nu\left(\dot\gamma_{i,j}\right)\left(\vartheta_{i,j}^u\Delta y_j\right)^2}{V_{i,j}} - \frac{\nu\left(\dot\gamma_{i+1,j}\right)\left(\vartheta_{i,j}^u\Delta y_j\right)^2}{V_{i+1,j}} - K_S^{\tau,x}(i,j) - K_N^{\tau,x}(i,j), \quad K_N^{\tau,x}(i,j) = \frac{\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta x_{i,j}^{xy}}{\Delta y_{i,j}^{xy}},$$

$$K_E^{\tau,x}(i,j) = \frac{\nu\left(\dot\gamma_{i+1,j}\right)\vartheta_{i,j}^u\vartheta_{i+1,j}^u\left(\Delta y_j\right)^2}{V_{i+1,j}}, \quad K_S^{\tau,x}(i,j) = K_N^{\tau,x}(i,j-1), \quad K_W^{\tau,x}(i,j) = K_E^{\tau,x}(i-1,j),$$

$$S_{\tau,x|_{i,j}}^{ib,\nu} = \vartheta_{i,j}^u\left(\Delta y_j\right)^2\left(\frac{\nu\left(\dot\gamma_{i+1,j}\right)\left(\vartheta_{i,j}^u-\vartheta_{i+1,j}^u\right)}{V_{i+1,j}}u_{i+1,j}^{ib} - \frac{\nu\left(\dot\gamma_{i,j}\right)\left(\vartheta_{i-1,j}^u-\vartheta_{i,j}^u\right)}{V_{i,j}}u_{i,j}^{ib}\right), i = \overline{2,N-2}, j = \overline{2,M-1};$$

$$\int\limits_{\Gamma_{i,j}^{v}} \nu(\dot\gamma)\left[\frac{\partial v}{\partial x}\overrightarrow{e_x}+\frac{\partial v}{\partial y}\overrightarrow{e_y}\right]\cdot\vec{n}\,dS \approx K_W^{\tau,y}(i,j)v_{i-1,j}+K_E^{\tau,y}(i,j)v_{i+1,j}+K_P^{\tau,y}(i,j)v_{i,j}+$$

$$+K_S^{\tau,y}(i,j)v_{i,j-1}+K_N^{\tau,y}(i,j)v_{i,j+1}+S_{\tau,y}^{ib,\nu}|_{i,j},$$

$$K_P^{\tau,y}(i,j)=-\frac{\nu(\dot\gamma_{i,j})\left(\vartheta_{i,j}^{v}\Delta x_i\right)^2}{V_{i,j}}-\frac{\nu(\dot\gamma_{i,j+1})\left(\vartheta_{i,j}^{v}\Delta x_i\right)^2}{V_{i,j+1}}-K_W^{\tau,y}(i,j)-K_E^{\tau,y}(i,j),\quad K_E^{\tau,y}(i,j)=\frac{\nu\left(\dot\gamma_{i,j}^{xy}\right)\Delta y_{i,j}^{xy}}{\Delta x_{i,j}^{xy}},$$

$$K_N^{\tau,y}(i,j)=\frac{\nu(\dot\gamma_{i,j+1})\vartheta_{i,j}^{v}\vartheta_{i,j+1}^{v}(\Delta x_i)^2}{V_{i,j+1}},\quad K_S^{\tau,y}(i,j)=K_N^{\tau,y}(i,j-1),\quad K_W^{\tau,y}(i,j)=K_E^{\tau,y}(i-1,j),$$

$$S_{\tau,y}^{ib,\nu}|_{i,j}=\vartheta_{i,j}^{v}(\Delta x_i)^2\left(\frac{\nu(\dot\gamma_{i,j+1})\left(\vartheta_{i,j}^{v}-\vartheta_{i,j+1}^{v}\right)}{V_{i,j+1}}v_{i,j+1}^{ib}-\frac{\nu(\dot\gamma_{i,j})\left(\vartheta_{i,j-1}^{v}-\vartheta_{i,j}^{v}\right)}{V_{i,j}}v_{i,j}^{ib}\right),\ i=\overline{2,N-1},\ j=\overline{2,M-2}.$$

Отметим, что для элементов матриц используются следующие обозначения: в строке соответствующей ячейке с номером (i,j) элемент с индексом P стоит на диагонали, элемент с индексом W находится в столбце с номером контрольного объема, граничащего с $\Omega_{i,j}$ с запада, и т. д.

Вблизи погруженных границ эти формулы необходимо модифицировать по аналогии со случаем ньютоновской жидкости, описанным в [5]. Перечислим все изменения в формулах.

При $\vartheta_{i,j+1}^{u}=0$ из $K_P^{\tau,x}(i,j)$ нужно вычесть $\frac{s_{i,j}^{ib,\tau,x}}{\vartheta_{i,j}^{u}\Delta y_j}$, а к $S_{\tau,x}^{ib,\nu}|_{i,j}$ – прибавить $\frac{s_{i,j}^{ib,\tau,x}u\left(x_{i,j}^{ib}\right)}{\vartheta_{i,j}^{u}\Delta y_j}$, где $s_{i,j}^{ib,\tau,x}=2\nu\left(\dot\gamma_{i,j}^{xy}\right)\left(\Delta x_{i,j}^{ib,e}+\Delta x_{i+1,j}^{ib,w}\right)$. Аналогично при $\vartheta_{i,j-1}^{u}=0$ из $K_P^{\tau,x}(i,j)$ нужно вычесть $\frac{s_{i,j-1}^{ib,\tau,x}}{\vartheta_{i,j}^{u}\Delta y_j}$, к $S_{\tau,x}^{ib,\nu}|_{i,j}$ – прибавить $\frac{s_{i,j-1}^{ib,\tau,x}u\left(x_{i,j}^{ib}\right)}{\vartheta_{i,j}^{u}\Delta y_j}$; при $\vartheta_{i+1,j}^{v}=0$ из $K_P^{\tau,y}(i,j)$ нужно вычесть $\frac{s_{i,j}^{ib,\tau,y}}{\vartheta_{i,j}^{v}\Delta x_i}$, где $s_{i,j}^{ib,\tau,y}=2\nu\left(\dot\gamma_{i,j}^{xy}\right)\left(\Delta y_{i,j}^{ib,n}+\Delta y_{i,j+1}^{ib,s}\right)$, при этом к $S_{\tau,y}^{ib,\nu}|_{i,j}$ нужно прибавить $\frac{s_{i,j}^{ib,\tau,y}v\left(x_{i,j}^{ib},y_j\right)}{\vartheta_{i,j}^{v}\Delta x_i}$; соответственно при $\vartheta_{i-1,j}^{v}=0$ из $K_P^{\tau,y}(i,j)$ нужно вычесть $\frac{s_{i-1,j}^{ib,\tau,y}}{\vartheta_{i,j}^{v}\Delta x_i}$, а к $S_{\tau,y}^{ib,\nu}|_{i,j}$ – прибавить $\frac{s_{i-1,j}^{ib,\tau,y}v\left(x_{i,j}^{ib},y_j\right)}{\vartheta_{i,j}^{v}\Delta x_i}$. Тогда в силу граничных условий в $S_{\tau,x}^{ib,\nu}$ ненулевые элементы будут только на границах Γ_1, Γ_2, Γ_3: $S_{\tau,x}^{ib,\nu}|_{i,1}=\frac{\nu\left(\dot\gamma_{i,0}^{xy}\right)V_\infty(\Delta x_i+\Delta x_{i+1})}{\vartheta_{i,1}^{u}\Delta y_1}$, $S_{\tau,x}^{ib,\nu}|_{i,M}=\frac{\nu\left(\dot\gamma_{i,M}^{xy}\right)V_\infty(\Delta x_i+\Delta x_{i+1})}{\vartheta_{i,M}^{u}\Delta y_M}$, $i=\overline{2,N-2}$; $S_{\tau,x}^{ib,\nu}|_{1,j}=\frac{\nu(\dot\gamma_{1,j})V_\infty\left(\vartheta_{1,j}^{u}\Delta y_j\right)^2}{V_{1,j}}$ при $j=\overline{2,M-1}$; $S_{\tau,x}^{ib,\nu}|_{1,1}=\frac{\nu(\dot\gamma_{1,1})V_\infty\left(\vartheta_{1,1}^{u}\Delta y_1\right)^2}{V_{1,1}}+\frac{\nu\left(\dot\gamma_{1,0}^{xy}\right)V_\infty(\Delta x_1+\Delta x_2)}{\vartheta_{1,1}^{u}\Delta y_1}$; $S_{\tau,x}^{ib,\nu}|_{1,M}=\frac{\nu(\dot\gamma_{1,M})V_\infty\left(\vartheta_{1,M}^{u}\Delta y_M\right)^2}{V_{1,M}}+\frac{\nu\left(\dot\gamma_{1,M}^{xy}\right)V_\infty(\Delta x_1+\Delta x_2)}{\vartheta_{1,M}^{u}\Delta y_M}$), кроме того на Γ_4 из $K_P^{\tau,y}(N,j)$ не нужно вычитать $s_{N,j}^{ib,\tau,y}$, к $K_P^{\tau,x}(N-1,j)$ надо добавить $\frac{\nu(\dot\gamma_{N,j})\left(\vartheta_{N-1,j}^{u}\Delta y_j\right)^2}{V_{N,j}}$. Значения $\Delta x_{i,j}^{ib,e}$, $\Delta x_{i+1,j}^{ib,w}$, $\Delta y_{i,j}^{ib,n}$, $\Delta y_{i,j+1}^{ib,s}$ зависят от типа усеченной ячейки.

Формулы для вычисления компонент гидродинамической силы $\vec{F}=\left(F_{xa},F_{ya}\right)$ строятся по аналогии с представленными в [5] для случая ньютоновской жидкости с той лишь разницей, что вместо постоянной вязкости ν в них используется $\nu(\dot\gamma)$:

$$F_{xa}^{h}=\sum_{\text{Cut-cells }\Omega_{i,j}^{ib}}\left(\vartheta_{i-1,j}^{u}-\vartheta_{i,j}^{u}\right)\Delta y_j\left(p_{i,j}-\nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial u}{\partial x}\right|_{i,j}\right)-\nu\left(\dot\gamma_{i,j}\right)\mathrm{Quad}_{i,j}^{ib}\left(\frac{\partial u}{\partial y}\overrightarrow{e_y}\cdot\vec{n}\right),$$

$$F_{ya}^{h}=\sum_{\text{Cut-cells }\Omega_{i,j}^{ib}}-\nu\left(\dot\gamma_{i,j}\right)\mathrm{Quad}_{i,j}^{ib}\left(\frac{\partial v}{\partial x}\overrightarrow{e_x}\cdot\vec{n}\right)+\left(\vartheta_{i,j-1}^{v}-\vartheta_{i,j}^{v}\right)\Delta x_i\left(p_{i,j}-\nu\left(\dot\gamma_{i,j}\right)\left.\frac{\partial v}{\partial y}\right|_{i,j}\right).$$

В данных выражениях квадратуры от давления и нормальных напряжений вычислены по формуле прямоугольников с учетом того, что эти слагаемые являются константами на усеченных ячейках; в результате одна и та же формула оказывается верной для всех типов усеченных ячеек. Наоборот, квадратуры от касательных напряжений, обозначенные как $\mathrm{Quad}_{i,j}^{ib}$, зависят от типа усеченной ячейки и вычисляются по формуле трапеций. Формулы для расчета $\mathrm{Quad}_{i,j}^{ib}$ представлены в [5].

Интегрирование по времени получающейся после LS-STAG-дискретизации по пространству дифференциально-алгебраической системы производится при помощи метода, основанного на схеме предиктор-корректор первого порядка. Этот метод состоит из двух шагов.

Шаг предиктора приводит к решению независимых разностных аналогов уравнений Гельмгольца для прогнозов скоростей $\tilde{u}$ и $\tilde{v}$ в момент времени $t_{n+1} = (n+1)\Delta t$:

$$\frac{M^x(\tilde{u}-u^n)}{\Delta t} + C^x\left[\overline{u}^n\right]u^n + S_x^{ib,c,n} - (D^x)^T P^n - K^{\tau,x}\tilde{u} - S_{\tau,x}^{ib,\nu} - F_u^{d,n} = 0,$$
$$\frac{M^y(\tilde{v}-v^n)}{\Delta t} + C^y\left[\overline{v}^n\right]v^n + S_y^{ib,c,n} - (D^y)^T P^n - K^{\tau,y}\tilde{v} - S_{\tau,y}^{ib,\nu} - F_v^{d,n} = 0.$$

Здесь P^n – вектор с компонентами $p_{i,j}^n$; u^n и v^n – векторы, компонентами которых являются значения скоростей $u_{i,j}^n$ и $v_{i,j}^n$ соответственно. Подчеркнем, что элементы матриц $K^{\tau,x}$ и $K^{\tau,y}$ зависят от шага по времени: при вычислении $\nu(\dot{\gamma})$ используются значения $\dot{\gamma}$ с n-го шага по времени. Таким образом, системы с шага предиктора являются линейными.

Шаг корректора приводит к решению разностного аналога уравнения Пуассона для функции давления $\Phi = \Delta t(P^{n+1} - P^n)$:

$$A\Phi = D^x\tilde{u} + D^y\tilde{v} + \overline{U}^{ib,n+1}.$$

Здесь $A = -D^x(M^x)^{-1}(D^x)^T - D^y(M^y)^{-1}(D^y)^T$, $A \in M(\mathbb{R})_{G\times G}$.

Затем определяются скорости и давление в момент времени t_{n+1}:

$$u^{n+1} = \tilde{u} + (M^x)^{-1}(D^x)^T\Phi, \quad v^{n+1} = \tilde{v} + (M^y)^{-1}(D^y)^T\Phi, \quad P^{n+1} = P^n + (\Phi/\Delta t).$$

4. Вычислительные эксперименты

Рассматривается степенная жидкость [20], поскольку для этого случая в литературе можно найти данные расчетов других исследователей для верификации результатов [25-27]. Для степенной жидкости зависимость вязкости от интенсивности тензора скоростей деформации имеет вид

$$\nu = m\dot{\gamma}^{n-1}.$$

Мера консистентности m в данном случае выражается через характерные параметры течения следующим образом:

$$m = \frac{\rho V_\infty^{2-n} D^n}{\mathrm{Re}}$$

При проведении расчетов безразмерные стационарные аэродинамические коэффициенты лобового сопротивления C_{xa} и подъемной силы C_{ya} получаются осреднением по большому промежутку времени нестационарных нагрузок:

$$C_{xa}(t) = \frac{2F_{xa}}{\rho V_\infty^2 D}, \quad C_{ya}(t) = \frac{2F_{ya}}{\rho V_\infty^2 D}.$$

Здесь ародинамические силы F_{xa} и F_{ya} вычисляются по формулам, полученным в предыдущем разделе.

Расчеты проводились при значениях числа Рейнольса $\mathrm{Re} < 50$ на неравномерной прямоугольной сетке 120×148 с шагом по времени $\Delta t = 0{,}01$. В окрестности погруженной

границы используется блок равномерной сетки с шагом $h = D/16$. Индекс течения n изменялся в диапазоне от 0,2 до 1,4.

На рис. 4 представлены картины обтекания кругового профиля потоком рассматриваемой степенной жидкости при $\mathrm{Re} = 40$ и различных значениях индекса течения. Видно, что длина зоны возвратного течения за круговым профилем увеличивается вместе с ростом значения индекса течения n, что согласуется с выводами исследований [25-27].

Сравнение значений безразмерного стационарного аэродинамического коэффициента лобового сопротивления C_{xa}, рассчитанных в случае обтекания неподвижного профиля при $\mathrm{Re} = \{0{,}1;\ 1;\ 10;\ 40\}$, с результатами работ [25-27] представлено в табл. 3 и 4. При всех рассматриваемых значениях числа Рейнольдса Re и индекса течения n относительная погрешность при сравнении рассчитанных значений C_{xa} с расчетами других исследователей не превосходит 10 %.

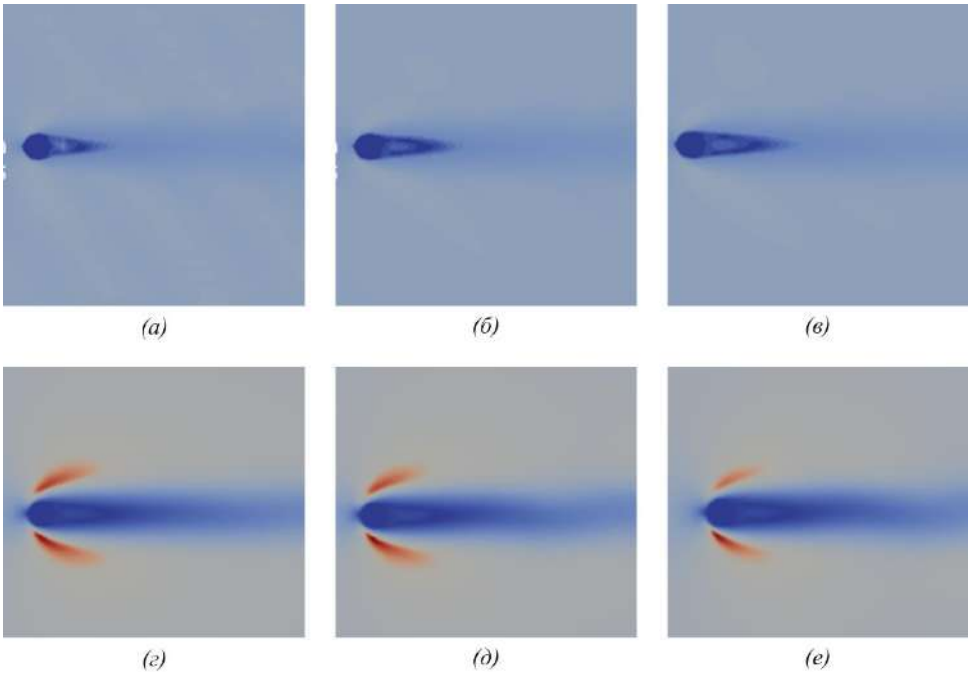


Рис. 4. Картины обтекания кругового профиля потоком степенной жидкости при $Re = 40$:
(а) $n = 0{,}4$; (б) $n = 0{,}6$; (в) $n = 0{,}8$; (г) $n = 1{,}0$; (д) $n = 1{,}2$; (е) $n = 1{,}4$.
Fig. 4. Power-law flow past circular airfoil at $Re = 40$:
(а) $n = 0{,}4$; (б) $n = 0{,}6$; (в) $n = 0{,}8$; (г) $n = 1{,}0$; (д) $n = 1{,}2$; (е) $n = 1{,}4$.

Табл. 3. Сравнение значений C_{xa}, рассчитанных при $Re > 15$, с результатами работ [25-27].
Table 3. Comparison of computed values of C_{xa} with the results of studies [25-27] at Re > 15.

Re	20					40					
n	0,6	0,8	1,0	1,2	1,4	0,2	0,4	0,6	0,8	1,0	1,4
Данная работа	1,88	1,89	1,90	1,92	1,94	1,11	1,25	1,35	1,37	1,45	1,50
Расчет [25]	1,97	2,04	2,01	2,03	2,05	–	–	–	–	–	–
Расчет [26]	1,96	1,99	2,05	–	2,10	–	–	1,28	1,42	1,53	1,54
Расчет [27]	–	–	–	–	–	1,14	1,24	1,35	1,44	1,51	–

Табл. 4. Сравнение значений C_{xa}, рассчитанных при $Re < 15$, с результатами работы [27].
Table 4. Comparison of computed values of C_{xa} with the results of study [27] at Re < 15.

n		0,2	0,4	0,6	0,8	1,0
$\mathrm{Re} = 0{,}1$	Данная работа	260,82	220,94	160,56	90,82	59,10
	Расчет [27]	270,54	231,32	164,25	100,85	61,13
$\mathrm{Re} = 1$	Данная работа	25,45	21,31	16,83	12,41	11,10
	Расчет [27]	27,06	23,31	17,47	13,25	10,57
$\mathrm{Re} = 10$	Данная работа	2,99	2,98	2,93	2,87	2,73
	Расчет [27]	3,16	3,03	2,93	2,86	2,80

5. Заключение

Разработана модификация метода погруженных границ LS-STAG для моделирования течений неньютоновских вязких жидкостей. Построенный численный метод может использоваться как для вязкопластичных жидкостей, так и для обобщенных ньютоновских жидкостей, не обладающих пределом текучести. Для вязкопластичных жидкостей рассмотрены модели Офоли-Моргана-Штеффе, Мизрахи-Берка, Кассо и Гершеля-Балкли, используемые с моделями регуляризации Берковьера-Энгельмана и Папанастасио, а для жидкостей, не обладающих пределом текучести, модели Эллиса, Кросса, Карро, Язуды (Карро-Язуды), а также степенная модель Оствальда-де Веле. Для верификации разработанного и реализованного в авторском программном комплексе [28] численного метода использовалась хорошо исследованная задача об обтекании неподвижного кругового профиля потоком степенной жидкости при различных значениях числа Рейнольдса и индекса течения. Полученные результаты хорошо согласуются с известными в литературе расчетными данными других исследователей. В дальнейшем планируется обобщить разработанную модификацию метода для расчета неньютоновских вязких жидкостей на случай подвижных погруженных границ.

Список литературы / References

[1]. Bird R.B., Armstrong R.C., Hassager O. Dynamics of polymeric liquids. V.1. Fluid mechanics. John Wiley & Sons, 1987. 649 p.

[2]. Owens R.G., Phillips T.N. Computational Rheology. Imperial College Press, 2002. 417 p.

[3]. Mittal R., Iaccarino G. Immersed boundary methods. Annu. Rev. Fluid Mech, 2005, vol. 37, pp. 239–261. DOI: 10.1146/annurev.fluid.37.061903.175743

[4]. Osher S. J., Fedkiw R. Level set methods and dynamic implicit surfaces. Springer, 2003. 286 p.

[5]. Cheny Y., Botella O. The LS-STAG method: A new immersed boundary/level-set method for the computation of incompressible viscous flows in complex moving geometries with good conservation properties. J. Comput. Phys., 2010, vol. 229, pp. 1043-1076. DOI: 10.1016/j.jcp.2009.10.007

[6]. Marchevsky I.K., Puzikova V.V. The modified LS-STAG method application for planar viscoelastic flow computation in a 4:1 contraction channel. Herald of the Bauman Moscow State Technical University. Natural Sciences, 2021, № 3, pp. 46-63. DOI: 10.18698/1812-3368-2021-3-46-63

[7]. Puzikova V.V. Simulation of viscoelastic flow past circular airfoil by using the modified LS-STAG immersed boundary method. J. Phys.: Conf. Ser. 2019. Vol. 1348, № 1. P. 1-8. DOI: 10.1088/1742-6596/1348/1/012093

[8]. Пузикова В.В. Модификация метода погруженных границ LS-STAG для моделирования течений вязкоупругих жидкостей. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 71-84. / Puzikova V. The LS-STAG Immersed Boundary Method Modification for Viscoelastic Flow Computations. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 1, 2017, pp. 71-84 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)- 5

[9]. Kraposhin M., Kuzmina K., Marchevsky I., Puzikova V. Study of OpenFOAM® efficiency for solving Fluid–Structure Interaction problems. OpenFOAM®: Selected papers of the 11th Workshop. 2019. P. 465 - 479.
[10]. Marchevsky I.K., Puzikova V.V. Numerical simulation in coupled hydroelastic problems by using the LS-STAG immersed boundary method. In: Gutschmidt S., Hewett J., Sellier M. (eds) IUTAM Symposium on Recent Advances in Moving Boundary Problems in Mechanics. IUTAM Bookseries, vol 34. Springer, Cham. 2019. P. 133-145. DOI: 10.1007/978-3-030-13720-5_12
[11]. Marchevsky I.K., Puzikova V.V. Numerical simulation of wind turbine rotors autorotation by using the modified LS-STAG immersed boundary method. International Journal of Rotating Machinery. 2017. V. 2017. P. 1-7. DOI: 10.1155/2017/6418108
[12]. Ofoli R.Y., Morgan R.G., Steffe J.F. A generalized rheological model for inelastic fluid foods. J. Texture Studies. 1987. Vol. 18, pp. 213-230.
[13]. Mitsoulis E. Flows of viscoplastic materials: models and computations. Rheology Reviews. 2007, pp. 135 - 178.
[14]. Rao M.A. Rheology of Fluid and Semisolid Foods: Principles and Applications. Springer, 2007. 433 p.
[15]. Bercovier M., Engelman M., A finite-element method for incompressible non-Newtonian flows. J. Comput. Phys. 1980. Vol. 36, pp. 313-326.
[16]. Papanastasiou T.C. Flow of materials with yield. J. Rheol. 1987. Vol. 31, pp. 385-404.
[17]. Mizrahi S., Berk S. Flow behavior of concentrated orange juice: mathematical treatment. J. Texture Studies. 1972. Vol. 3, pp. 69-79.
[18]. Herschel W.H., Bulkley R. Konsistenzmessungen von Gummi-Benzol-Loesungen. Kolloid Z. 1926. Vol. 39, pp. 291-300.
[19]. Bingham E.C. Fluidity and Plasticity. N.-Y.: McGraw-Hill, 1922. 440 p.
[20]. de Waele A. Viscometry and plastometry // Oil Color Chem. Assoc. J. 1923. Vol. 6, pp. 33-69.
[21]. Barnes H.A., Hutton J.E., Walters K. An introduction to rheology. Elsevier, 1993. 199 p.
[22]. Cross M. Rheology of non-Newtonian fluids: a new flow equation for pseudo-plastic systems. J. Colloid. 1958. Vol. 20, pp. 417-437.
[23]. Carreau P. Rheological equations from molecular network theories. Trans. Soc.Rheol. 1972. Vol. 16, pp. 99-127.
[24]. Yasuda R.A.K., Cohen R. Shear-flow properties of concentrated-solutions of linear and star branched polystyrenes. Rheol. Acta. 1981. Vol. 20, pp. 163-178.
[25]. Sivakumar P., Bharti R.P., Chhabra R.P. Effect of power-law index on critical parameters for power-law flow across an unconfined circular cylinder // Chemical Engineering Science. 2006. V. 61, issue 18, pp. 6035-6046.
[26]. Bharti R.P., Chhabra R.P., Eswaran V. Steady flow of power law fluids across a circular cylinder. The Canadian Journal of Chemical Engineering. 2006. V. 84, pp. 406-421.
[27]. Panda S.K., Chhabra R.P. Laminar flow of power-law fluids past a rotating cylinder. Journal of Non-Newtonian Fluid Mechanics. 2010. V. 165, pp. 1442-1461.
[28]. Пузикова В. В. Параллельный программный комплекс LS-STAG_GNF для моделирования течений неньютоновских вязких жидкостей (обобщенных ньютоновских жидкостей); свидетельство о государственной регистрации программы для ЭВМ № 2023663406, зарегистрировано в Реестре программ для ЭВМ 16.06.23 / Puzikova V. V. Parallel software package LS-STAG_GNF for non-Newtonian viscous (generalized Newtonian) fluid simulation; Certificate of state registration of a computer program No. 2023663406, registered in the Register of Computer Programs on 06/16/23 (in Russian).

Информация об авторах / Information about authors

Илья Константинович МАРЧЕВСКИЙ – доктор физико-математических наук, доцент, профессор кафедры «Прикладная математика» МГТУ им. Н.Э. Баумана. Сфера научных интересов: вычислительная гидродинамика, вихревые методы, теория устойчивости, численные методы, высокопроизводительные вычисления, элементарная математика.

Ilia Konstantinovich MARCHEVSKY – Dr. Sci. (Phys.-Math.), Associate professor, Professor of Applied Mathematics department, Bauman Moscow State Technical University. Research interests:

computational fluid dynamics, vortex particle methods, theory of stability, numerical methods, high performance computing, elementary mathematics.

Валерия Валентиновна ПУЗИКОВА – кандидат физико-математических наук, эксперт по разработке программного обеспечения Отдела разработки высокопроизводительных библиотек компании YADRO. Сфера научных интересов: решатели и предобуславливатели СЛАУ, разработка прикладных математических программ, вычислительная гидродинамика, физические движки для AR/VR, высокопроизводительные вычисления, численные методы.

Valeria Valentinovna PUZIKOVA – Cand. Sci. (Phys.-Math.), Software Development Expert in High Performance Libraries Department, YADRO. Research interests: solvers and preconditioners for SLAE, applied mathematics software development, computational hydrodynamics, physics engines for AR/VR, high performance computations, numerical methods.