

ТРУДЫ

**ИНСТИТУТА СИСТЕМНОГО
ПРОГРАММИРОВАНИЯ РАН**

**PROCEEDINGS OF THE INSTITUTE
FOR SYSTEM PROGRAMMING OF THE RAS**

ISSN Print 2079-8156
Том 37 Выпуск 4 Часть 1

ISSN Online 2220-6426
Volume 37 Issue 1 Part 1

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2025

ИСП **РАН**

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:



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)

[Ластовенский 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 институт Technion (Хайфа, Израиль)

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/>

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)

[Andrew Voronkov](#), Dr. Sci. (Phys.–Math.), Professor, University of Manchester (Manchester, United Kingdom)

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>

С о д е р ж а н и е

Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений. <i>Бурдонов И.Б., Евтушенко Н.В., Косачев А.С., Пономаренко В.Н.</i>	7
Алгоритм оптимизации «черепаха и заяц» с адаптивной стратегией на основе взаимной информации для обнаружения сетевых вторжений. <i>Бхуванешвари Т., Руба Соундар К., Чандра Гуру Секар Р.</i>	31
Определение неточностей в работе некоторых специализированных цикловых оптимизаций в компиляторе LCC для архитектуры «Эльбрус». <i>Ермолицкий А.В., Левченко Д.Н., Нейман-заде М.И.</i>	51
Поиск мёртвых вызовов функций как дефектов исходного кода методами статического анализа. <i>Васильева В.И., Бородин А.Е., Волков А. Е.</i>	65
NOREC: компилятор специализированных регулярных выражений для проектирования программируемой и ресурсоэффективной аппаратной архитектуры. <i>Советов П.Н.</i>	79
Извлечение функциональности из бинарного кода. <i>Ильина А.А., Курмангалеев Ш.Ф.</i>	97
Обзор методов миграции программных интерфейсов приложений для объектно-ориентированных языков. <i>Чуркин Я.А., Мельник Д.М., Бучацкий Р.А.</i>	111
Реализация гибкой системы управления доступом в приложении. <i>Васильев Т.И.</i>	147
Управление MDM-проектами. <i>Кузнецов С.В., Кознов Д.В., Луцив Д.В.</i>	161
MaxSMT-решатель, поддерживающий режим портфолио. <i>Фомина В.В., Соболев В., Кознов Д.В.</i>	177
Локализация целей в реальном времени на БПЛА с лазерным дальномером на подвесе. <i>Саакян В.Р., Мелконян В.Г., Саргсян С.С.</i>	189
Именная морфология первого печатного памятника тверской карельской письменности. <i>Новак И.П.</i>	199
Управление батареей электромобиля на основе машинного обучения и ПЛИС. <i>Мерина Р.Д., Рам Р.С., Майкл Л.С.П.</i>	215

Применение компьютерного зрения для локализации позвонков на срединных сагиттальных срезах компьютерной томографии.
Теплякова А.Р., Шеринев Р.В. 233

Table of Contents

A distributed network model in which hosts can perform message switching functions. Burdonov I.B., Yevtushenko N.V., Kossatchev A.S., Ponomarenko V.N.	7
Rabbit and Tortoise optimization algorithm with mutual information based adaptive strategy for network intrusion detection. <i>Bhuvaneswari T., Ruba Soundar K., Chandra Guru Sekar R.</i>	31
Detection of inaccuracies in some specialized loop optimizations in the LCC compiler for “Elbrus” architecture. <i>Ermolitsky A.V., Levchenko D.N., Neiman-zade M.I.</i>	51
Detection of Dead Function Calls as Source Code Defects through Static Analysis. Vasileva V.I., Borodin A.E., Volkov A.E.	65
HOREC: a specialized regular expression compiler for designing programmable and resource-efficient hardware architecture. <i>Sovietov P.N.</i>	79
Extraction of Functionality from Binary Code. <i>Ilin A.A., Kurmangaleev Sh.F.</i>	97
Survey of Application Program Interface Migration Methods for object-oriented languages. <i>Churkin Y.A., Melnik D.M., Buchatskiy R.A.</i>	111
Implementation of a flexible access control system in the application. <i>Vasilev T.I.</i>	147
Managing Master Data Implementations. <i>Kuznetsov S.V., Koznov D.V., Luciv D.V.</i>	161
MaxSMT solver with portfolio mode support. <i>Fomina V.V., Sobol V., Koznov D.V.</i>	177
Real-Time Target Localization Using Gimbaled Laser on UAVs. <i>Sahakyan V.R., Melkonyan V.G., Sargsyan S.S.</i>	189
Nominal morphology of the earliest printed piece of Tver Karelian written language heritage. <i>Novak I.P.</i>	199
Machine Learning-Enabled Battery Management System on FPGA for Electric Vehicles. <i>Merina R.D., Ram R.S., Micheal L.C.P.</i>	215
Application of Computer Vision for Localization of Vertebrae on Midsagittal Computed Tomography Slices. <i>Teplyakova A.R., Shershnev R.V.</i>	233



Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений

¹ И.Б. Бурдонов, ORCID: 0000-0001-9539-7853 <igor@ispras.ru>

^{1,2} Н.В. Евтушенко, ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>

¹ А.С. Косачев, ORCID: 0000-0001-5316-3813 <kos@ispras.ru>

¹ В.Н. Пономаренко, ORCID: 0009-0002-2387-2760 <vera@ispras.ru>

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

² Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20.

Аннотация. В работе строится абстрактная модель распределенной сети, содержащей только хосты и коммутаторы, которая позволяет оценить классы задач, которые необходимо решать в такой сети, принимая во внимание, в том числе, нефункциональные параметры. Предполагается, что хосты предлагают пакеты определенных услуг (сервисов) и сообщения (запросы) между хостами пересылаются через промежуточные узлы по правилам коммутации. Правило определяет, каким соседним узлам пересылается принятое узлом сообщение в зависимости от того, откуда оно пришло, и от вектора параметров в его заголовке. Соответственно, настройка узлов определяет множество путей от хоста к хосту, по которым будут пересылаться пакеты. Ситуация моделируется с использованием графа, вершинами которого являются хосты и коммутаторы, а ребра соответствуют физическим связям между ними. Обычно предполагается, что в такой сети хосты только принимают, обрабатывают и посылают информацию другим хостам, но не занимаются коммутацией сообщений, эта функция возлагается на другие узлы – коммутаторы, но мы предполагаем, что при современных технологиях хост также может выполнять функции коммутации сообщений, то есть такой хост (как и коммутатор) содержит систему правил коммутации, указывающих, куда отправляется полученное сообщение, если почему-либо данный хост не может обработать данный запрос/сообщение. Предлагается модель сети, в которой функцию коммутации сообщений выполняет не только каждый коммутатор, но и каждый хост. Обсуждаются проблемы, связанные с нефункциональными параметрами распределенной сети, а именно, *достижимость/недостижимость* хостов, *заикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*. Обсуждается возможность оптимизации рассматриваемых параметров сети на основе использования информации об услугах/сервисах, представляемых каждым из хостов, и алгоритмы самонастройки распределенной сети, оптимизирующие параметры сети, передачу сообщений по настроенной сети и инкрементальную (повторную частичную) настройку сети, не нарушающую функционирование сети.

Ключевые слова: распределенная сеть; хосты; коммутаторы; достижимость; заикливание; перегрузка; масштабируемость.

Для цитирования: Бурдонов И.Б., Евтушенко Н.В., Косачев А.С., Пономаренко В.Н. Модель распределённой сети, в которой хосты могут выполнять функцию коммутации сообщений. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 7–30. DOI: 10.15514/ISPRAS–2025–37(4)–1.

A Distributed Network Model in which Hosts Can Perform Message Switching Functions

¹ I.B. Burdonov ORCID: 0000-0001-9539-7853 <igor@ispras.ru>

^{1,2} N.V. Yevtushenko ORCID: 0000-0002-4006-1161 <evtushenko@ispras.ru>

¹ A.S. Kossatchev ORCID: 0000-0001-5316-3813 <kos@ispras.ru>

¹ V. N. Ponomarenko, ORCID: 0009-0002-2387-2760 <vera@ispras.ru>

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

² National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. In the paper, an abstract model of a distributed network containing only hosts and switches is developed; this model allows us to estimate some problems that need to be solved in such a network, taking into account, among other things, non-functional parameters. It is assumed that hosts offer packages of certain services, and messages (requests) between hosts are forwarded via intermediate nodes according to switching rules. A rule determines which neighboring nodes a message received by a node is forwarded to, depending on where the message came from and the parameter vector in its header. Accordingly, the configuration of nodes determines the set of paths from host to host along which packets will be forwarded. The situation is modeled using a graph of physical connections, the vertices of which are hosts and switches, and the edges correspond to the physical connections between them. It is usually assumed that only hosts in such a network receive, process, and send information to other hosts, but do not switch messages. This function is assigned to other nodes, switches. However, we assume that with modern technologies, in some cases a host can perform the switching functions, i.e. such a host (like a switch) contains a system of switching rules that specify where the received message is sent if for some reason the host cannot process the obtained request/message. A network model is proposed in which the message switching function is performed not only by each switch, but also by each host. The paper discusses the problems associated with non-functional parameters of a distributed network, namely, reachability/unreachability of hosts, message looping, network overload with messages, and non-scalability. Optimization issues of the network parameters are discussed based on the set of services provided by each host. In addition, we discuss self-tuning algorithms for a distributed network that optimize network parameters, message transmission over the configured network and incremental (repeated partial) self-tuning algorithms when changing network parameters, in particular, its topology that does not disrupt network operation.

Keywords: distributed network; hosts; switches; reachability; looping; overload; scalability.

For citation: Burdonov I.B., Yevtushenko N.V., Kossatchev A.S., Ponomarenko V. N. A distributed network model in which hosts can perform message switching functions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 7-30 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-1.

1. Введение

В распределенных сетях (distributed networks) отсутствует центральное управление для пересылки и обработки информации, т.е. каждый элемент такой сети обладает полной автономией [1]. В этой области есть большое количество публикаций, и в большинстве работ [см. например, 1, 2] рассматриваются сети, используемые для передачи данных в специальных условиях и для специальных целей, содержащие ряд различных специальных устройств. В некоторых случаях узлы современной сети делятся на два класса: хосты и коммутаторы, и в качестве примера таких сетей можно рассматривать программно-конфигурируемые сети (*software-defined networking*, SDN) [3-8]. В настоящей работе мы строим абстрактную модель распределенной сети, содержащей только хосты и коммутаторы, которая позволяет оценить классы задач, которые необходимо решать в такой сети, принимая во внимание, в том числе, нефункциональные параметры. Мы предполагаем, что хосты предлагают пакеты определенных услуг (сервисов) и сообщения (запросы) между хостами

пересылаются через промежуточные узлы (коммутаторы). В данной работе, так же, как и в большинстве программно-конфигурируемых сетей коммутатор работает по правилам коммутации. Правило определяет, каким соседним узлам пересылается принятое коммутатором сообщение в зависимости от того, откуда оно пришло, и от вектора параметров в его заголовке [6-8]. Соответственно, настройка коммутаторов определяет множество путей от хоста к хосту, по которым будут пересылаться пакеты. Ситуация моделируется с использованием графа физических связей, вершинами которого являются хосты и коммутаторы, а рёбра соответствуют физическим связям между ними. Иногда предполагается, что каждый хост может быть соединен только с одним коммутатором [6-8]. В общем случае хосты в такой сети принимают, обрабатывают и посылают информацию другим хостам, но мы предполагаем, что при современных технологиях хост может выполнять также функции коммутации сообщений, т.е. такой хост (как и коммутатор) содержит систему правил коммутации, указывающих, куда отправить полученный запрос/сообщение, если почему-либо данный хост не может его обработать. В программно-конфигурируемых сетях настройка правил коммутации обычно осуществляется специальными компонентами сети, например, SDN-контроллерами, но в ряде случаев возможна и самонастройка коммутаторов, в зависимости от передаваемых запросов, а также инкрементальная (повторная и частичная) настройка при изменении параметров сети, в частности, её топологии.

Структура работы следующая. В разделе 2 обсуждаются проблемы, связанные с нефункциональными параметрами распределенной сети: *достижимость/недостижимость* хостов, *заикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*. В разделе 3 описывается предлагаемая модель распределённой сети, а в разделе 4 рассматривается передача сообщений по сети известным хостам-получателям. В разделе 5 обсуждается возможность оптимизации рассматриваемых параметров сети на основе использования информации об услугах/сервисах, предоставляемых каждым из хостов. В разделе 6 рассматривается настройка распределенной сети, а в разделе 7 – проблемы инкрементальной (повторной и частичной) настройки функционирующей сети (когда в ней продолжают циркулировать сообщения) при различных изменениях параметров сети. В разделах 3-7 описана идея предлагаемых алгоритмов; сами алгоритмы вынесены в приложение.

2. Четыре проблемы распределённых сетей

В качестве модели распределённой сети (далее, просто сети) рассматривается граф, в котором вершины – это хосты и коммутаторы, а рёбра – каналы связи, по которым передаются сообщения. В настоящей статье мы будем считать каналы связи дуплексными, т.е. сообщения могут передавать по ребру в обоих направлениях, поэтому граф считается неориентированным. Ребру $\{a, b\}$ соответствуют две ориентированные дуги ab и ba , вершины a и b называются *соседними*. Считается, что в графе нет кратных рёбер и петель. В этом случае путь в графе однозначно определяется как последовательность вершин. Если граф не связный, то сеть, фактически, распадается на независимые подсети, соответствующие компонентам связности, которые функционируют независимо друг от друга. В дальнейшем будем считать, что граф связный. Мы будем использовать слова «вершина» (графа) и «узел» (сети) как синонимы.

Под хостом понимается узел сети, который может обрабатывать информацию, например, выполнять вычислительные функции. Именно хост генерирует сообщения, передаваемые далее по сети, и именно хост является конечным получателем сообщений.

Коммутаторы выполняют функцию коммутации сообщений: приняв сообщение от соседней вершины a , коммутатор s пересылает его другому (или тому же самому) своему соседу b . В этом случае будем называть соседа a *предшественником*, а соседа b *преемником*

коммутатора s . Мы предполагаем, что само сообщение при этом не меняется. Коммутатор работает по правилам коммутации, которые хранятся в его памяти. Каждое правило коммутатора s в общем случае имеет вид: $(p: a, s, b)$, где a – сосед-предшественник, b – сосед-преемник, p – параметры коммутации, т.е. та часть параметров сообщения, которая определяет выбор того или иного правила. Если сообщение принято коммутатором s от предшественника a и параметры сообщения соответствуют параметрам коммутации p в правиле вида $(p: a, s, b)$, то это правило срабатывает, и сообщение будет направлено соседу b . Если есть несколько правил, отличающихся друг от друга только преемником; предполагается, что все такие правила срабатывают. Тем самым, если срабатывают несколько правил, то сообщение *клонировается*, и каждый клон посылается соответствующему преемнику. При клонировании сообщение (его клоны) могут получить несколько хостов, поэтому, если сообщение предназначено не для всех этих хостов, например, для одного хоста или определённой группы хостов, то остальные хосты, получив такое сообщение, должны его игнорировать.

Замечание. В правиле коммутации $(p: a, s, b)$ указаны вершины графа, точнее, идентификаторы этих вершин. В реализации можно было бы обойтись без идентификаторов вершин, если считать, что все рёбра, инцидентные вершине s , линейно упорядочены, например, пронумерованы от 1 до $\deg(s)$, где $\deg(s)$ степень вершины s (число её соседей). Когда вершина s получает сообщение от предшественника a , ей должен автоматически сообщаться номер $i(a, s)$ её ребра $\{a, s\}$, по которому получено сообщение. Когда вершина s посылает сообщение преемнику b , она указывает номер $i(s, b)$ её ребра $\{s, b\}$, по которому посылается сообщение. Правило коммутации вместо $(p: a, s, b)$ имело бы вид $(p: i(a, s), s, i(s, b))$. С другой стороны, идентификатор вершины-хоста необходим как параметр сообщений, которые посылаются конкретному хосту-получателю. Поэтому для удобства изложения мы будем считать, что заданы идентификаторы вершин, как описано выше, а правила коммутации имеют вид $(p: a, s, b)$. В то же время, учитывая такую реализацию с номерами рёбер, будем считать, что когда вершина s получает сообщение от предшественника a , идентификатор предшественника a является не параметром сообщения, а ответным параметром оператора приёма сообщения в s . Когда вершина s посылает сообщение преемнику b , идентификатор преемника b является не параметром сообщения, а параметром оператора отправки сообщения.

Правила коммутации порождают пути в графе, по которым двигаются сообщения. Путь $a_1, a_2, \dots, a_n$, где a_1 и a_n хосты, а остальные вершины коммутаторы, порождается правилами $(p: a_1, a_2, a_3)$, $(p: a_2, a_3, a_4)$, ..., $(p: a_{n-2}, a_{n-1}, a_n)$.

С такой распределённой сетью связан ряд проблем: *недостижимость* хостов, *зацикливание* сообщений, *перегрузка* сети сообщениями, *немасштабируемость*.

Достижимость хостов означает, что для каждой упорядоченной пары хостов (a, b) правила коммутации позволяют переслать сообщение из хоста a в хост b . Соответственно, проблема недостижимости возникает, когда это требование не выполняется.

Зацикливание сообщений возникает тогда, когда правила коммутации определяют путь, проходящий дважды по одной и той же дуге. В этом случае отрезок пути, начинающийся и заканчивающийся этой дугой, является ориентированным циклом, по которому сообщение будет двигаться бесконечно. Путь, не проходящий дважды по одной дуге, называется *простым по дугам*. Таким образом, для того чтобы не было зацикливания, все порождаемые пути должны быть простыми по дугам.

Перегрузка сети может возникнуть из-за клонирования сообщений, когда в сети циркулирует слишком много сообщений. Наилучшим решением было бы вообще отказаться от излишнего клонирования, когда число клонов больше числа хостов, которым сообщение предназначено.

Масштабируемость означает, что суммарное (во всех коммутаторах) число правил коммутации не более, чем линейно, зависит от числа вершин графа. Иными словами, число

правил в одном коммутаторе ограничено числом, не зависящим от числа вершин графа. Например, при отсутствии параметров коммутации, когда правило коммутации имеет вид $(: a, s, b)$, число правил в коммутаторе s не превосходит $\deg(s)^2$, и не зависит от общего числа вершин в графе. В идеале в масштабируемой распределённой сети добавление в граф новых вершин и рёбер требует изменения/добавления правил только в тех «старых» коммутаторах, которым инцидентны новые рёбра. На практике это обычно не так, но в целом число изменяемых правил не должно зависеть от общего числа вершин графа.

На самом деле, проблемы, описанные выше, тесно связаны между собой, причём некоторые решения одних проблем усугубляют другие проблемы. Проиллюстрируем это простейшими примерами решения этих проблем.

Достижимость можно обеспечить просто клонированием сообщения в коммутаторе так, чтобы оно пересылалось всем соседям, кроме того, от которого получено. Сообщение с гарантией достигнет нужного хоста-получателя (поскольку граф связан). Однако, во-первых, это может привести к заикливанию, если в графе есть цикл, а, во-вторых, и как следствие, к перегрузке сети.

Вместо этого можно посылать сообщение из хоста-отправителя по остовному дереву, ориентированному от корня, которым этот отправитель и будет. Сообщение будет клонироваться в точках разветвления дерева. Заикливания не будет, однако проблема перегрузки всё равно остаётся: вместо одного сообщения будет число сообщений, равное числу листовых вершин остовного дерева. Кроме того, в этом случае нам нужно различать сообщения, сгенерированные разными хостами-отправителями, т.е. идентификатор хоста-отправителя будет частью параметров коммутации, что приведёт к немасштабируемости сети.

Можно посылать сообщение по остовному дереву, ориентированному к корню, которым будет хост-получатель. Заикливания не будет, и сообщение не будет клонироваться, и поэтому не будет перегрузки. Но проблема немасштабируемости остаётся: идентификатор хоста-получателя будет частью параметров коммутации.

Наилучшим (из простейших) решений было бы выделение в графе ориентированного простого по дугам цикла, который содержал бы все хосты. Сообщение пересылается по этому циклу до нужного хоста. Цикл $a_1, a_2, \dots, a_n, a_1$ порождается правилами коммутации $(: a_1, a_2, a_3)$, $(: a_2, a_3, a_4)$, ..., $(: a_{n-2}, a_{n-1}, a_n)$, $(: a_{n-1}, a_n, a_1)$, т.е. параметры коммутации отсутствуют. В этом случае можно решить все четыре проблемы.

Но для этого нужно, чтобы хосты могли выполнять, кроме вычислительных функций, также и функцию коммутации. Во многих распределённых сетях, например, в программно-конфигурируемых сетях (SDN), как правило, эти функции не совмещаются: вершина графа – либо хост, либо коммутатор, но не то и другое вместе. Более того, часто требуется, чтобы хост был терминальной вершиной графа, т.е. был подключён только к одному коммутатору.

Это приводит к целому ряду проблем. Например, если в графе есть мост, с каждой стороны которого есть несколько хостов, то невозможно одновременно решить проблемы недостижимости, перегрузки и немасштабируемости. Для того чтобы обеспечить достижимость, либо сообщения, проходящие по мосту в одном направлении, должны клонироваться после моста, чтобы попасть во все хосты с другой стороны моста, либо мы должны различать в правилах коммутации сообщения, направляемые разным хостам. В первом случае у нас будет перегрузка: создаются «лишние» клоны сообщений для хостов, которым они не предназначены. А во втором случае будет немасштабируемость: идентификатор хоста-получателя становится частью параметров коммутации. Эти проблемы пытаются решить, допуская клонирование, но в ограниченном масштабе, когда проблема перегрузки решается лишь частично.

3. Модель распределённой сети

Авторами предлагается модель распределённой сети, в которой не только коммутаторы, но и хосты выполняют функцию коммутации сообщений. Как будет показано далее, это помогает решить описанные выше проблемы вполне удовлетворительно. В дальнейшем будем считать, что все вершины графа делятся на хосты, которые выполняют как вычислительные функции, так и функцию коммутации сообщений, и коммутаторы, которые выполняют только функцию коммутации.

В этом и следующем разделах мы рассмотрим функционирование такой сети, когда выполнена её настройка, т.е., прежде всего, установлены нужные правила коммутации в вершинах графа, а также инициализированы другие переменные вершин, необходимые для передачи сообщений или для инкрементальной (дополнительной частичной) настройки. При этом мы будем иметь в виду, что в процессе такого функционирования граф может изменяться и сеть может перенастраиваться. В разделе 6 мы рассмотрим самонастройку сети, а в разделе 7 – различные виды изменений сети и требуемую при этих изменений перенастройку сети.

Будем считать, что каждое сообщение в сети имеет тип (имя) и набор параметров. Мы будем опираться на следующее требование неизменности сообщения: при пересылке сообщения по сети меняться может только тип сообщения, параметры остаются неизменными. При этом число типов сообщения ограничено и фиксировано; в предлагаемых ниже алгоритмах используется 9 типов сообщений.

Прежде всего, заметим, что ориентированный простой по дугам цикл, содержащий все хосты, о котором шла речь в предыдущем разделе, всегда существует в связном графе. Таким циклом является цикл обхода остовного дерева графа, например, по алгоритму Тэрри [9], который проходит каждое ребро два раза, но по одному разу в каждом направлении, т.е. по одному разу по каждой дуге (на рис. 1.а обход остовного дерева показан синим цветом).

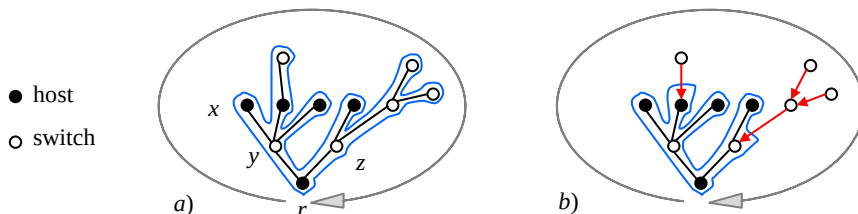


Рис. 1. а) Обход остовного дерева, б) Обход дерева хостов.
Fig. 1. а) Spanning tree traversal, б) Host tree traversal.

Для данного корневого дерева его обход, начиная с корня, по алгоритму Тэрри однозначно определяется линейным порядком соседей вершины, заданным для каждой вершины: соседи вершины проходятся алгоритмом в этом порядке. Родителем некорневой вершины x будем называть такую вершину y , что дуга xy последняя на пути от корня до вершины x ; в алгоритме Тэрри по этой дуге мы первый раз попадаем в вершину x . Родителем корня r будем условно называть его последнего соседа z в линейном порядке соседей корня; в алгоритме Тэрри дуга zr проходится последней.

В дальнейшем будем считать, что корень остовного дерева является хостом. Если остовное дерево содержит листовые коммутаторы (не хосты), то они, очевидно, являются «лишними» при коммутации сообщений, и их можно удалить. Минимальное поддереву остовного дерева с тем же корнем, которое содержит все хосты, будем для краткости называть *деревом хостов*, нужно только иметь в виду, что кроме хостов оно содержит также коммутаторы, необходимые для связности дерева. Эквивалентное определение: *деревом хостов* называется поддереву остовного дерева с тем же корнем, которое содержит все хосты, и все его листья

являются хостами. Соответственно, вместо цикла обхода остоного дерева будем использовать цикл обхода дерева хостов, который для краткости будем называть *циклом хостов*. Там, где это не приводит к двусмысленности, корень дерева хостов будем для краткости называть просто *корнем*.

Дерево хостов строится процедурой «Удаление "лишних" коммутаторов». Первоначально дерево хостов совпадает с остовным деревом. Если коммутатор является листом дерева, он удаляется вместе с единственным инцидентным ему ребром дерева. Процедура повторяется до тех пор, пока все листья дерева не будут хостами. Это и будет дерево хостов, обход которого является циклом хостов (на рис. 1.b обход хостов показан синим цветом).

Рассмотрим ту часть остоного дерева, которая не вошла в дерево хостов. Она представляет собой лес корневых деревьев, все вершины которых, кроме, быть может, их корней, являются коммутаторами и не лежат на дереве хостов, а их корни лежат на дереве хостов. Будем называть эти деревья *деревьями коммутаторов*. Два дерева коммутаторов не имеют общих вершин. Для передачи сообщений по сети деревья коммутаторов не нужны, так как все хосты (а только они генерируют сообщения) находятся на цикле хостов, и сгенерированные сообщения двигаются по циклу хостов и не попадают в коммутаторы, не являющиеся корнями деревьев коммутаторов. Однако лес деревьев коммутаторов может быть нужен, когда происходит изменение графа сети. Например, к сети может быть добавлен новый хост, подсоединяемый к коммутатору на дереве коммутаторов. Поэтому будем считать, что правила коммутации порождают, кроме цикла хостов, лес деревьев коммутаторов, ориентированных к своим корням (на рис. 1.b отмечены красным цветом).

Мы будем считать, что в настроенной сети в каждой вершине инициализированы следующие переменные: *Self* – собственный идентификатор вершины, *Host* – отметка хоста, *Rules* – правила коммутации, *Root* – отметка корня дерева хостов, где «отметка» – булевская переменная.

4. Передача сообщений известному хосту-получателю

Для отправки сообщения известному хосту-получателю используется сообщение типа **MessageToHost** с параметрами: *sender* – идентификатор хоста-отправителя, *recipient* – идентификатор хоста, которому предназначено сообщение, *parameters* – параметры сообщения, прозрачные для коммутации сообщений. Сообщение двигается по циклу хостов до хоста-получателя с идентификатором *recipient*. Получатель, приняв такое сообщение, обрабатывает его, не посылая дальше.

Что произойдёт, если в сети нет хоста с указанным в сообщении идентификатором получателя? Заметим, что это может произойти не только в результате ошибки в хосте-отправителе, который генерирует сообщение с таким идентификатором получателя, но также из-за изменения сети, например, когда хост удаляется из сети и после соответствующей перенастройки сети удаляется из цикла хостов. В этом случае сообщение будет бесконечно циркулировать по циклу хостов.

Для того чтобы это предотвратить, предлагается такое решение. Когда фиксируется, что сообщение прошло полный цикл хостов, не найдя получателя, оно удаляется, а отправителю посылается сообщение с отрицательным ответом. Фиксация может быть выполнена корнем, контролирующим проход сообщения через него. Однако цикл хостов является простым по дугам путём, который в общем случае не проходит дважды по одной дуге, но может несколько раз проходить через одну и ту же вершину (т.е. не обязательно является вершинно-простым путём), в том числе корень. Поэтому корень сначала фиксирует проход сообщения **MessageToHost** по дуге, ведущей от родителя корня в корень, и меняет его тип на **RootMessageToHost**. Когда последнее сообщение проходит по той же дуге, ведущей от родителя корня в корень, корень удаляет сообщение и посылает отправителю сообщение **MessageToHost** с отрицательным ответом. Пометив штрихом параметры этого

отрицательного ответа как функцию параметров исходного сообщения, имеем: $sender' = root$, $recipient' = sender$, $parameters' = parameters$, где $root$ идентификатор корня.

Этот отрицательный ответ также может не найти своего получателя $recipient' = sender$ из-за того, что хост-отправитель исходного сообщения неверно указал свой идентификатор $sender$ или этот отправитель удалён из сети и (после перенастройки сети) из цикла хостов. Тогда по общему правилу ответное сообщение (отправитель – корень), дважды пройдя по дуге, ведущей от родителя корня в корень, будет удалено, и корень пошлёт отрицательный ответ на этот отрицательный ответ, получателем которого будет он сам: $recipient'' = sender' = root$. Получив это сообщение, предназначенное корню, корень окончательно удалит его.

Схема передачи по сети сообщения известному получателю наглядно изображена на рис. 2, где двойная стрелка соответствует передаче сообщения по циклу хостов через вершины, не меняющие тип сообщения, цветной кружок означает хост-отправитель или хост-получатель, а белый кружок означает проход по дуге от родителя корня в корень. При передаче сообщений по этой схеме может случиться неправильное поведение: сообщение (ответное сообщение с отрицательным ответом от корня) может быть удалено, хотя в сети есть получатель этого сообщения (получатель ответного сообщения, т.е. отправитель исходного сообщения). Однако это может случиться только тогда, когда меняется корень, т.е. удаляется корень, сеть перенастраивается, и корнем становится другой хост. Удаление получателей и отправителей сообщения, отличных от корня, не приводит к такому неправильному поведению.

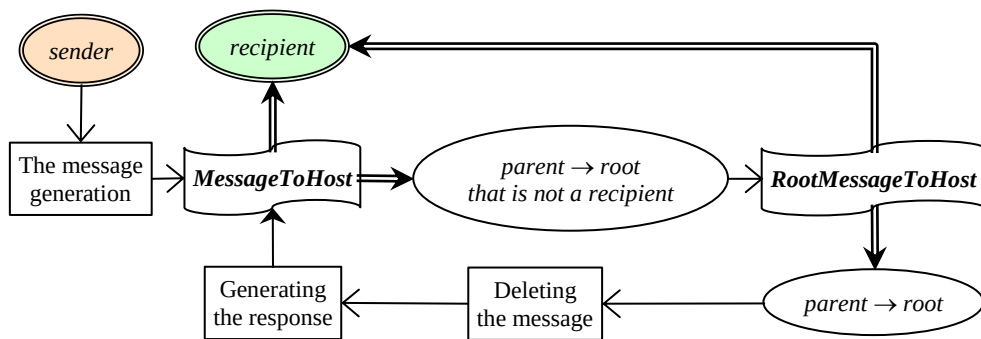


Рис. 2. Схема передачи по сети сообщения известному получателю.

Fig. 2. The network transmission diagram when sending a message to the known recipient.

Для генерации сообщения **MessageToHost** конкретному указанному хосту используется процедура **SendMessageToHost** с параметрами: *recipient*, *parameters*. Процедура возвращает **false**, если некому послать сообщение, т.е. в хосте нет правил коммутации (сеть состоит из одной изолированной вершины). В противном случае возвращается **true** и посылается по циклу хостов нужное сообщение: с типом **MessageToHost**, если данный хост не корень, или с типом **RootMessageToHost** в противном случае.

5. Услуги

Другой аспект функционирования распределённой сети связан с тем, что сообщение может быть предназначено не для какого-то заранее известного хоста, а «кому-нибудь», кто может это сообщение обработать. Такое сообщение можно понимать как запрос на некие вычислительные действия с параметрами, содержащимися в сообщении. Это аналогично локальному вызову той или иной процедуры внутри одного хоста с той разницей, что вызывающий процесс и вызываемая процедура находятся в разных хостах. При этом, вообще говоря, всё равно в каком хосте эта вызываемая процедура находится, лишь бы она там была. Коммутация сообщений должна позволять «найти» такой хост.

Мы предлагаем моделировать такой запрос понятием *услуги*. Хост, который может оказывать данную услугу (в хосте реализована данная услуга), будем называть *целевым хостом* для этой услуги. В сообщении запроса услуги указывается имя услуги и набор параметров, а в каждом хосте имеется множество имён услуг, которые в этом хосте реализованы, т.е. услуги, которые могут быть оказаны этим хостом. Сообщение передаётся по циклу хостов. Когда сообщение приходит в некоторый хост, то хост проверяет, может ли он оказать запрашиваемую услугу, т.е. является ли целевым хостом для этой услуги. Если не может, хост выполняет функцию коммутации, пересылая сообщение дальше согласно правилам коммутации. Если может, этот хост становится конечным получателем сообщения, сообщение дальше не пересылается, а обрабатывается в этом хосте, т.е. хост выполняет запрос, оказывая запрашиваемую услугу.

Однако такое решение имеет следующий недостаток: сообщения с данным именем услуги, отправляемые из одного и того же хоста-отправителя, будут приниматься одним и тем же хостом-получателем, а именно, ближайшим по циклу целевым хостом. Это может привести к тому, что одни хосты будут перегружены обработкой сообщений, а другие, которые тоже могли бы оказывать какие-то услуги, будут недогружены.

Для решения этой проблемы можно разрешить хосту, который мог бы оказать запрашиваемую услугу, но сейчас слишком «занят» другой работой, не принимать сообщение на обработку, а пересылать его дальше по правилам коммутации. Тем самым, сообщение будет «крутиться» по циклу до тех пор, пока не попадёт в какой-нибудь освободившийся хост, который может оказать запрашиваемую услугу.

Для того чтобы в такой распределённой сети не возникало заикливание, нужно, чтобы каждая запрашиваемая услуга была реализована в каком-нибудь хосте. Иначе сообщение с именем этой услуги будет бесконечно двигаться по циклу, не находя нужного хоста. Для того чтобы обойти это ограничение предлагается решение аналогичное решению, рассмотренному в предыдущем разделе для сообщений указанному хосту.

Вводятся три типа сообщений: **Message**, **RootMessage** и **WaitingMessage**. Они имеют параметры: *sender* – идентификатор хоста-отправителя, сгенерировавшего сообщение, *service* – имя запрашиваемой услуги, *parameters* – параметры сообщения, прозрачные для коммутации сообщений.

Сначала сообщение, как правило, посылается с типом **Message**. Если оно приходит в корень, который не является целевым хостом, по дуге, ведущей от родителя корня в корень, сообщение пересылается дальше по циклу с типом **RootMessage**. Если сообщение **Message** или **RootMessage** приходит в целевой хост, но сейчас он «занят», хост пересылает сообщение дальше уже с типом **WaitingMessage**. Корень, не являющийся целевым хостом, получая сообщение **WaitingMessage** по дуге, ведущей от родителя корня в корень, пересылает его дальше опять с типом **Message**. Последнее сделано для того, что «поймать» удаление из цикла хостов того занятого целевого хоста, который сменил тип сообщения с **Message** на **WaitingMessage**. Если в цикле ещё остаются целевые хосты, сообщение, проходя через целевой хост, либо принимается им, если хост «свободен», либо посылается дальше опять с типом **WaitingMessage**, если хост сейчас «занят». Если сообщение **RootMessage** проходит по дуге, ведущей от родителя корня в корень, оно удаляется, а корень генерирует и посылает сообщение типа **MessageToHost** хосту-отправителю с отрицательным ответом.

Тем самым, в цикле может «крутиться» только сообщение с запросом услуги, для которой в цикле хостов есть целевые хосты. Но это будет не бесконечно, а до тех пор, пока какой-нибудь целевой хост не освободится и не примет это сообщение, или пока из цикла хостов не будут удалены все целевые хосты при перестройке сети. В то же время число проходов таким сообщением по циклу хостов не ограничено.

Схема передачи по сети сообщения с запросом услуги наглядно изображена на рис. 3, где двойная стрелка соответствует передаче сообщения по циклу хостов через вершины, не являющиеся целевыми хостами или корнем, цветной кружок означает хост-отправитель или

целевой хост, а белый кружок означает проход по дуге от родителя корня в корень. Многоточие показывает, что передача по сети отрицательного ответа от корня отправителю исходного сообщения выполняется, естественно, по общим правилам для сообщения типа **MessageToHost** (как на рис. 2).

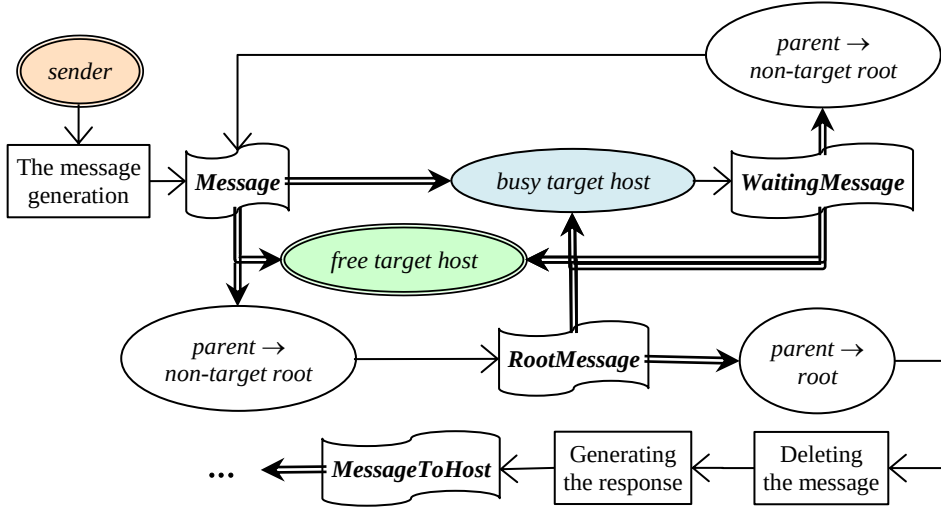


Рис. 3. Схема передачи по сети сообщения запроса услуги.
Fig. 3. The network transmission diagram when sending a service request message.

Предполагается, что когда хост принимает сообщение, оказывая запрашиваемую услугу, то после этого он может, если нужно, сам послать ответ хосту-отправителю, сгенерировавшему это сообщение с результатами оказания услуги. Заметим, что это не всегда нужно. Например, когда требуется некоторая цепочка услуг или более сложная программа, отдельные части которой понимаются как услуги, которые могут быть реализованы в разных хостах, то ответ, если нужно, будет послан хосту-отправителю только после выполнения всей программы. Для посылки ответа хосту также используется сообщение типа **MessageToHost**, в котором указывается в качестве получателя идентификатор хоста-отправителя исходного сообщения. Другой способ послать ответное сообщение мог бы быть основан на пути, проходимом сообщением. Если такой путь запоминать в самом сообщении как последовательность проходимых вершин, то ответ мог бы проходить этот путь просто в обратном направлении. Более того, этот путь можно было бы «спрямить», удалив циклы, т.е. в обратном направлении проходить вершинно-простой путь (не проходящий дважды через одну вершину). Однако, во-первых, такой способ требует особой динамической коммутации, происходящей не по статическим правилам коммутации, хранящимся в вершинах, а по информации из сообщения, и, во-вторых, это противоречит нашему предположению о том, что сообщения передаются по сети без изменения (кроме их типа). Поэтому далее способы передачи сообщений такого рода не рассматриваются.

Для генерации в хосте сообщения запроса удалённой услуги используется процедура **SendMessage** с параметрами: *service* – имя запрашиваемой услуги, *parameters* – параметры сообщения, прозрачные для коммутации сообщений. Процедура возвращает **false**, если некому послать сообщение с запросом услуги, т.е. в хосте нет правил коммутации (хост изолированная вершина). В противном случае возвращается **true** и посылается нужное сообщение запроса услуги: **Message** – если отправитель не целевой хост и не корень, **RootMessage** – если отправитель не целевой корень, **WaitingMessage** – если отправитель целевой хост. В последнем случае предполагается, что хост делает удалённый вызов потому, что сейчас «занят».

6. Настройка сети

Под настройкой сети понимается, прежде всего, установление правил коммутации. Например, программно-конфигурируемая сеть (SDN) основана на физическом разделении плоскости данных (уровень передачи сообщений, моделируемый графом с вершинами в хостах и коммутаторах) и плоскости управления сетью. На плоскости управления находится контроллер, который и выполняет настройку сети. Он связан с каждым коммутатором, которому «спускает» на уровень плоскости данных правила коммутации. Для этого на уровне контроллера должна быть известна вся топология сети (граф физических связей).

В настоящей статье мы предлагаем алгоритмы самонастройки сети, без использования специального контроллера и с минимально необходимой информацией, заранее заданной в вершинах графа. Эта предварительно заданная информация должна быть необходима и достаточна только для задания топологии сети как упорядоченного графа (графа, в котором рёбра, инцидентные вершине, линейно упорядочены). Достаточность означает, что по такой информации однозначно восстанавливается упорядоченный граф сети, а необходимость означает, что по упорядоченному графу однозначно определяется такая информация. Кроме этого, каждый хост должен знать, какие услуги в нём реализованы.

До настройки сети в каждой вершине графа должна быть заранее задана следующая информация:

- *Self* – идентификатор этой вершины;
- список *Neighbors* идентификаторов соседних вершин;
- *Host* – признак того, является вершина хостом (*Host = true*) или коммутатором (*Host = false*).

Кроме того, в хосте должно быть задано множество *Services* имён реализуемых им услуг.

В предлагаемой модели под настройкой понимается установление:

- 1) правил коммутации в узлах сети,
- 2) отметки корня дерева хостов.

Поясним эти установки.

- 1) Правила коммутации устанавливаются в каждом узле сети: как в коммутаторе, так и в хосте, поскольку каждый узел сети выполняет функцию коммутации сообщений. В вершине инициализируется список *Rules* правил коммутации.
- 2) В корне дерева хостов инициализируется переменная *Root := true*, в остальных хостах сети *Root := false*. Как описано выше, это нужно для предотвращения бесконечной циркуляции сообщений в сети.

Алгоритмы настройки сети строят дерево хостов и цикл хостов как обход этого дерева, а также лес деревьев коммутаторов. Сами алгоритмы самонастройки, а также алгоритмы генерации и передачи сообщений по настроенной сети формально описаны в приложении.

Все правила коммутации в вершине *s* имеют вид $(: a, s, b)$, то есть параметры коммутации отсутствуют, а средняя вершина одна и та же – *s*. Поэтому в приложении для краткости в каждой вершине *s* правило коммутации записывается не как $(: a, s, b)$, а как (a, b) .

В результате настройки в каждой вершине *s* цикла хостов список правил коммутации будет иметь «циклический» вид $(: a_0, a_1), (: a_1, a_2), \dots, (: a_{n-2}, a_{n-1}), (: a_{n-1}, a_0)$, где a_0 родитель вершины *s*. Кроме того, если список правил коммутации в вершине *s* имеет такой вид, а сообщение в вершину *s* приходит от соседа *b*, отличного от вершин $a_0, \dots, a_{n-1}$, оно пересылается вершине a_0 – родителю вершины *s* по подразумеваемому правилу (b, a_0) . Иными словами, список $(a_0, a_1), (a_1, a_2), \dots, (a_{n-1}, a_0)$ следует понимать как сокращённую запись списка $(a_0, a_1), (a_1, a_2), \dots, (a_{n-1}, a_0), (b_1, a_0), (b_2, a_0), \dots, (b_k, a_0)$, где $b_1, \dots, b_k$ все соседи вершины *s*, кроме соседей $a_0, \dots, a_{n-1}$. Такое правило умолчания мы применяем для упрощения

алгоритмов самонастройки сети в Приложении. Это правило умолчания используется при инкрементальной (повторной частичной) перенастройке сети, когда меняется граф сети. Для передачи сообщений по настроенной сети это правило умолчания не требуется, за исключением одного случая: мы допускаем посылку сообщения извне сети в сеть, в вершину b с указанием идентификатора предшественника a , не совпадающего с идентификаторами вершин сети, что можно понимать как посылку сообщения по дуге ab , не входящей в граф сети.

На рис. 4 показано, какие правила создаются для вершины в разных случаях. Чёрные линии изображают рёбра дерева хостов, красные – деревьев коммутаторов. Правило (a, b) в вершине s показано стрелкой, соединяющей две смежные дуги as и sb ; синие стрелки соответствуют циклу хостов, а красные стрелки – деревьям коммутаторов. Показаны правила как до (рис. 4 сверху), так и после применения правила умолчания (рис. 4 внизу).

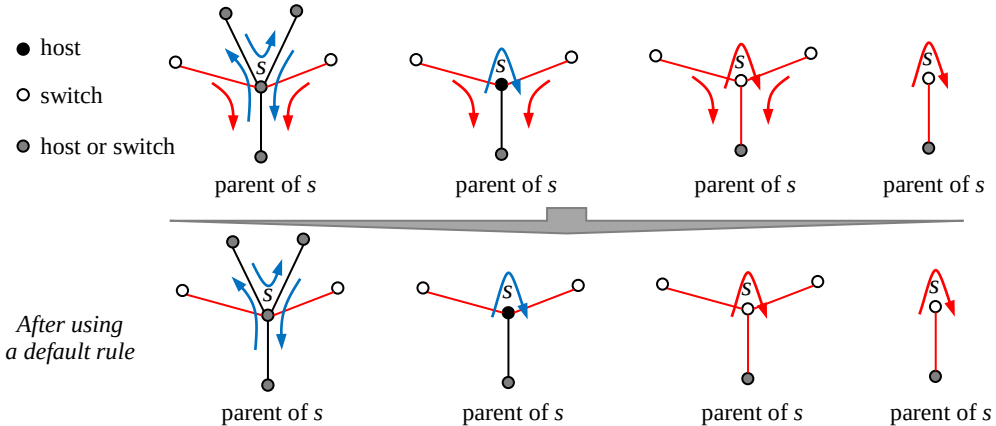


Рис. 4. Правила коммутации и правило умолчания.
 Fig. 4. Switching rules and a default rule.

Будем считать, что настройка инициируется сообщением **Start**, которое поступает в некоторый (произвольный) хост извне графа. Этот хост будем называть *инициатором*. Для удобства будем считать, что сообщение приходит по некоторому дополнительному ребру, соединяющему инициатор с внешним окружением, моделируемым дополнительной вершиной, соседней с инициатором, которую будем называть *внешним соседом* инициатора. Этому же внешнему соседу инициатор посылает ответ о выполнении этапа. В то же время будем предполагать, что внешний сосед не входит в список *Neighbors* соседей инициатора, так как этот список (по определению) содержит только вершины графа.

В предлагаемых алгоритмах во время настройки в сети циркулирует одно сообщение, которое имеет тип и не имеет параметров. При пересылке сообщения узлы сети могут менять только тип сообщения.

Замечание. Если разрешить изменяемые и неограниченные по размеру сообщения, то можно было бы с помощью сообщений сначала собрать всю информацию о сети. Для этого можно использовать сообщения, которые, обходя граф, собирали бы информацию о сети: как о её топологии, так и о распределении реализаций услуг по хостам сети. Получив такую информацию, инициатор (или его внешний сосед) мог бы произвести необходимую оптимизацию и составить описание, содержащее требуемую инициализацию переменных для каждого узла сети. После этого специальным сообщением, обходящим граф, каждому узлу сети доставлялась бы касающаяся его часть этого описания. Стоит отметить, что такой инициатор уже мало отличается от контроллера в SDN: только вместо прямой связи с каждым узлом сети ему приходится посылать сообщение по сети до этого узла. В настоящей статье мы не допускаем таких изменяемых сообщений, тем более, неограниченных по размеру.

Настройка сети может начать выполняться в любой момент времени, когда уже закончена предыдущая (если она была) настройка. Сначала неформально опишем алгоритм обхода остоного дерева по алгоритму Тэрри. Хост, получающий сообщение **Start**, становится инициатором и корнем дерева. Далее сообщение может иметь один из трёх типов: *прямое* сообщение **Forward** и два *ответных сообщения (ответа)* **Cancel** и **Back**. Сообщение **Forward** посылается по ещё не пройденной дуге графа. Если вершина b получает от соседа a сообщение **Forward** первый раз, то дуга ab становится дугой остоного дерева, ориентированного от корня; вершина a является родителем вершины b . На время настройки родителем корня (= инициатора) будем считать его внешнего соседа. Если вершина b получает от соседа a повторное сообщение **Forward**, то оно приходит по хорде ab остоного дерева (хорда – это ребро, не лежащее на дереве, оба конца которого лежат на дереве). В этом случае по этой хорде в обратном направлении ba посылается ответ **Cancel**. Иначе вершина b посылает сообщение **Forward** следующему после a соседу c , если дуга bc ещё не пройдена (по ней не посылалось сообщение **Forward**) и вершина c отлична от родителя. То же самое делается, если вершина b получает ответ **Cancel** или **Back** от соседа a . Все выходящие из вершины b дуги уже пройдены в том случае, когда следующий после a сосед c является родителем для вершины b . Тогда вершина b посылает своему родителю ответ **Back** по обратной дуге дерева хостов. По ходу дела происходит генерация правил коммутации сообщений для цикла обхода остоного дерева. В конце корень (= инициатор) замыкает цикл обхода, заменяя два сгенерированных правила (: *внешний сосед, инициатор, b*) и (: *c, инициатор, внешний сосед*) одним правилом (: *c, инициатор, b*). После окончания настройки сосед c становится родителем корня.

Для того, чтобы строить не цикл обхода остоного дерева, а цикл обхода дерева хостов, т.е. цикл хостов, применяется процедура «Удаление «лишних» коммутаторов». Для этого перед посылкой вершиной b ответа родителю по обратной дуге остоного дерева сначала проверяется, является ли вершина b хостом и, если нет, то имеются ли хосты в поддереве остоного дерева с корнем в вершине b . Такие хосты имеются, если либо (при использовании правила умолчания, рис. 4 внизу) в вершине b определено больше одного правила коммутации, либо (без использования правила умолчания, рис. 4 вверху) в этих правилах есть хотя бы два разных преемника. Если в поддереве остоного дерева с корнем в вершине b имеются хосты, вершина b посылает ответ **Back**, иначе посылается ответ **Cancel**.

Настройка заканчивается, когда корень (= инициатор) посылает своему внешнему соседу ответное сообщение **Back** (корень = инициатор является хостом, поэтому внешнему соседу корня = инициатора не может быть послан ответ **Cancel**).

7. Инкрементальная настройка сети

Под инкрементальной настройкой сети будем понимать дополнительную, чаще всего частичную перенастройку сети, необходимую при изменении сети. Такое изменение касается либо

- 1) распределения реализаций услуг по хостам сети, либо
- 2) топологии сети (изменение графа сети).

Каждое такое изменение можно представить как последовательность элементарных изменений. Поэтому мы будем рассматривать только элементарные изменения и соответствующие им инкрементальные настройки сети.

- 1) При изменении распределения реализаций услуг по хостам сети элементарными изменениями являются:
 - 1.1 добавление реализации одной услуги в один хост,
 - 1.2 удаление реализации одной услуги из одного хоста.

2) При изменении топологии сети элементарными изменениями являются:

- 2.1 изменение упорядочивания графа,
- 2.2 добавление одного ребра,
- 2.3 удаление одного ребра, не нарушающее связность графа,
- 2.4 добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа,
- 2.5 добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа,
- 2.6 удаление одного терминального коммутатора и инцидентного ему ребра,
- 2.7 удаление одного терминального хоста и инцидентного ему ребра.

Примером более сложного изменения топологии сети является удаление хоста и всех инцидентных ему рёбер. Ему соответствует последовательность удалений одного ребра (2.3) для всех инцидентных хосту рёбер, кроме одного, а затем удаление одного терминального хоста и инцидентного ему ребра (2.7).

В отличие от предыдущих разделов здесь мы предлагаем только идеи алгоритмов настройки, по которым легко можно разработать сами алгоритмы инкрементальной настройки, мы не приводим эти алгоритмы в Приложении. В некоторых случаях перенастройка сети из-за какого-то изменения может быть выполнена эффективнее, если её делать «сразу», а не как последовательность перенастроек сети по элементарным изменениям. Но это уже вопрос оптимизации (и усложнения) алгоритмов инкрементальной настройки. Кроме того, мы рассматриваем только такую инкрементальную настройку, которая необходима для поддержания функциональности сети, то есть возможности передавать сообщения от любого хоста-отправителя любому известному хосту или запрашивать любую услугу. Если перенастройка полезна для оптимизации сети (например, уменьшения длины цикла хостов), то мы только отмечаем этот факт, но не предлагаем алгоритмы такой перенастройки.

В ряде случаев во время инкрементальной настройки нужно знать, лежит ли вершина на цикле хостов или нет. У нас нет соответствующей отметки (булевой переменной) в вершинах графа, но она и не нужна. Вершина лежит на цикле хостов тогда и только тогда, когда она является хостом или это коммутатор, и в нём либо определено больше одного правила коммутации (при использовании правила умолчания, рис. 4 внизу), либо в этих правилах есть хотя бы два разных преемника (без использования правила умолчания, рис. 4 вверху).

Инкрементальная настройка сети, как правило, не нарушает функциональности сети. Если во время настройки имеется сообщение указанному хосту (**MessageToHost** или **RootMessageToHost**), оно будет доставлено получателю, если в сообщении верно указан его идентификатор, и получатель не удалён из сети. Иначе сообщение с отрицательным ответом (**MessageToHost** или **RootMessageToHost**), будет доставлено отправителю исходного сообщения, если в сообщении верно указан идентификатор отправителя, и отправитель не удалён из сети. Если во время настройки имеется сообщение с запросом услуги (**Message**, **RootMessage** или **WaitingMessage**), оно будет доставлено целевому хосту, если в сообщении верно указано имя услуги и в сети есть целевые хосты для этой услуги. Иначе сообщение с отрицательным ответом (**MessageToHost** или **RootMessageToHost**), будет доставлено отправителю исходного сообщения, если в сообщении верно указан идентификатор отправителя, и отправитель не удалён из сети. Исключением является случай, когда меняется корень дерева хостов: «старый» корень удаляется, и корнем становится другой хост. В этом случае сообщение может не дойти до адресата, даже если такой адресат есть в сети.

7.1 Изменение распределения реализаций услуг по хостам

7.1.1 Добавление реализации одной услуги в один хост

При добавлении реализации услуги с именем *service* в некоторый хост нужно выполнить в этом хосте $Services := Services \cup \{service\}$. Больше никаких действий не требуется.

7.1.2 Удаление реализации одной услуги из одного хоста

При удалении реализации услуги с именем *service* из некоторого хоста нужно выполнить в этом хосте $Services := Services \setminus \{service\}$. Больше никаких действий не требуется. Заметим, что хост, в котором не остаётся реализованных услуг, отличается от коммутатора, поскольку, хотя он не может оказывать услуг, но может генерировать сообщения с запросом услуг, и, кроме того, может быть получателем сообщений указанному хосту (*MessageToHost* или *RootMessageToHost*).

7.2 Изменение топологии сети

7.2.1 Изменение упорядочивания графа

Другое упорядочивание графа, т.е. определение других линейных порядков соседей каждой вершины, приводит к построению другого остовного дерева при настройке сети и, как следствие, других дерева и цикла хостов и леса деревьев коммутаторов. Для того чтобы задать новое упорядочивание графа, нужно в каждой вершине, в которой изменился линейный порядок соседей, установить новый список *Neighbors* идентификаторов соседних вершин. Однако для сохранения функциональности сети перенастройка не требуется. В то же время полная перенастройка сети могла бы быть полезна в целях оптимизации: уменьшения длины цикла хостов.

7.2.2 Добавление одного ребра

Для сохранения функциональности сети перенастройка не требуется. В то же время перенастройка сети могла бы быть полезна в целях оптимизации: уменьшения длины цикла хостов. Для подготовки к следующей настройке при добавлении ребра $\{a, b\}$ в список *Neighbors* в вершине *a* добавляется идентификатор соседа *b*, а в список *Neighbors* в вершине *b* добавляется идентификатор соседа *a*.

7.2.3 Удаление одного ребра, не нарушающее связность графа

Для подготовки к следующей настройке при удалении ребра $\{a, b\}$ из списка *Neighbors* в вершине *a* удаляется идентификатор соседа *b*, а из списка *Neighbors* в вершине *b* удаляется идентификатор соседа *a*.

Если удаляемое ребро является хордой остовного дерева, перенастройка не требуется (даже для оптимизации).

Если удаляемое ребро лежит на дереве коммутаторов, его удаление нарушает связность этого дерева (рис. 5). Если это ребро $\{a, b\}$, где вершина *a* является родителем для вершины *b*, то одна из этих компонент связности является поддеревом остовного дерева с корнем в вершине *b*, а другая компонента содержит цикл хостов. Поскольку граф остаётся связным, должно быть ребро $\{c, d\}$, соединяющее вершину *c* из первой компоненты с вершиной *d* из второй компоненты. Требуется изменить правила коммутации в вершинах пути по (неориентированному) дереву коммутаторов из вершины *b* в вершину *c*, а также в вершинах *a* и *d*. Однако с учётом правила умолчания правила в вершинах *a* и *d* можно не менять, так как в них не изменились родители по соответствующим деревьям (дерева хостов для вершины *a* и дерева коммутаторов для вершины *d*).

Если удаляемое ребро лежит на дереве хостов, его удаление нарушает связность этого дерева (рис. 6). Если это ребро $\{a, b\}$, где вершина a является родителем для вершины b , то одна из этих компонент связности является поддеревом остоного дерева с корнем в вершине b , а другая компонента содержит корень дерева хостов. Поскольку граф остаётся связным, должно быть ребро $\{c, d\}$, соединяющее вершину c из первой компоненты с вершиной d из второй компоненты. Требуется изменить правила коммутации в вершинах пути по (неориентированному) остоному дереву из вершины b в вершину c и в вершинах пути по (неориентированному) остоному дереву из вершины d до цикла хостов, а также в вершине a .

7.2.4 Добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа

Для подготовки к следующей настройке при добавлении коммутатора a и ребра $\{a, b\}$ создаётся список $Neighbors = \{b\}$ в коммутаторе a и в список $Neighbors$ в вершине b добавляется идентификатор соседа a . Для сохранения функциональности сети перенастройка не требуется. Однако при дальнейших изменениях топологии сети во время соответствующей перенастройки сети нужно учитывать добавляемые коммутатор и ребро. Поскольку при добавлении коммутатора a только с одним инцидентным ему ребром $\{a, b\}$, коммутатор a будет терминальной вершиной, он может входить только в дерево коммутаторов. Правила коммутации устанавливаются в добавляемом коммутаторе a и меняются в другом конце b добавляемого ребра (рис. 7). Однако с учётом правила умолчания для коммутации сообщений достаточно только в вершине a установить правило коммутации $\{ : b, a, b \}$.

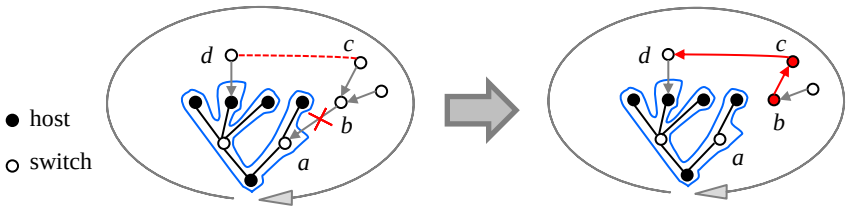


Рис. 5. Удаление ребра на дереве коммутаторов.
Fig. 5. Deleting an edge of a switch tree.

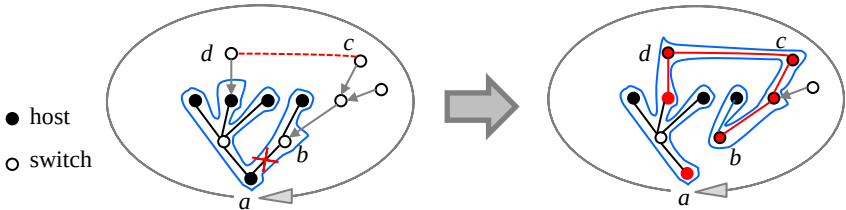


Рис. 6. Удаление ребра на дереве хостов.
Fig. 6. Deleting an edge of a host tree.

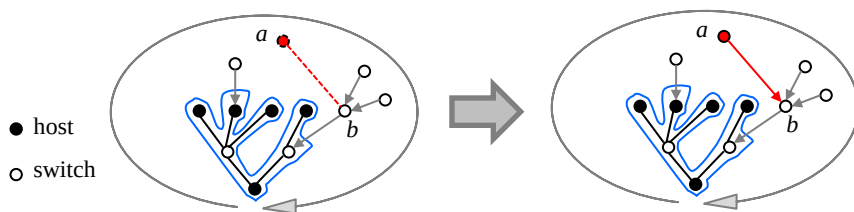


Рис. 7. Добавление одного коммутатора и одного ребра, соединяющего его со «старой» вершиной графа.

Fig. 7. Adding one host and one edge connecting that connects this host with the “old” graph node.

7.2.5 Добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа

Когда добавляется хост a и ребро $\{a, b\}$, соединяющее его со «старой» вершиной b , нужно создать список $Neighbors = \{b\}$ в хосте a , в список $Neighbors$ в вершине b добавить идентификатор соседа a , и добавить хост a в цикл хостов. Правила коммутации устанавливаются в добавляемом хосте a и меняются в вершине b ; кроме того, если вершина b не лежит на цикле хостов, то она является некорневой вершиной некоторого дерева коммутаторов, и нужно поменять правила коммутации на всём пути от вершины b до корня этого дерева (рис. 8).

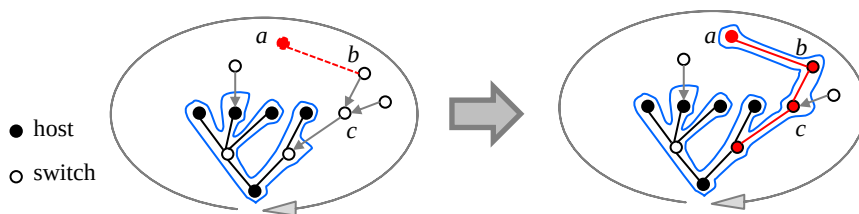


Рис. 8. Добавление одного хоста и одного ребра, соединяющего его со «старой» вершиной графа.

Fig. 8. Adding one switch and one edge connecting that connects this switch with the “old” graph node.

7.2.6 Удаление одного терминального коммутатора и инцидентного ему ребра

Связность графа не нарушается, перенастройка не требуется при использовании правила умолчания. Если правило умолчания не используется, при удалении терминального коммутатора a и ребра $\{a, b\}$, нужно в вершине b удалить правило $(: a, b, c)$. Для подготовки к следующей настройке из списка $Neighbors$ в вершине b удаляется идентификатор соседа a .

7.2.7 Удаление одного терминального хоста и инцидентного ему ребра

При удалении терминального хоста a и (единственного) инцидентного ему ребра $\{a, b\}$ связность графа не нарушается. Для подготовки к следующей настройке из списка $Neighbors$ в вершине b удаляется идентификатор соседа a . Удаляемый хост нужно удалить из цикла хостов. Поскольку хост терминальный, он является корнем или листовой вершиной дерева хостов. С учётом правила умолчания достаточно изменить правила коммутации в вершине b (рис. 9). Кроме того, если удаляется корень дерева хостов, нужно отметить в качестве корня другой (всё равно какой) хост.

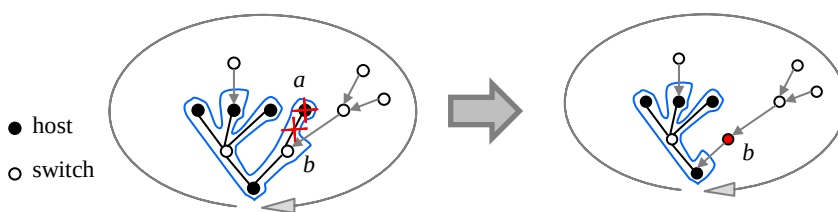


Рис. 9. Удаление одного терминального хоста и инцидентного ему ребра.

Fig. 9. Deleting a terminal host with an incident edge.

8. Заключение

У решения, предложенного в данной статье, есть один существенный недостаток: путь, который должно пройти сообщение, прежде чем «найдёт» нужный хост, может оказаться слишком длинным. Например, если в сети для некоторой услуги есть только один целевой хост a (в котором реализована эта услуга), то сообщение с запросом этой услуги, сгенерированное хостом b , следующим после хоста a в цикле хостов, пройдёт весь цикл хостов, кроме коммутаторов между a и b .

Для того чтобы устранить этот недостаток, можно было бы строить не общий цикл всех хостов, а циклы целевых хостов – по одному для каждой услуги в предположении, что таких циклов существенно меньше. Соответственно, имя услуги было бы параметром коммутации. Но такое решение плохо справляется с проблемой масштабируемости, если число услуг можно считать достаточно большим и, более того, если это число имеет тенденцию к росту. Для масштабируемого решения этой проблемы можно воспользоваться подходящей кластеризацией услуг. Предполагается, что этому будет посвящена наша следующая статья на эту тему.

Список литературы / References

- [1]. М.С.Э. Танненбаум. Распределенные системы, Москва, ДМК Пресс, 2021.
- [2]. Г.И. Радченко. Распределенные вычислительные системы. Челябинск, 2012.
- [3]. Sezer, S, Scott-Hayward, S, Chouhan P.K., Fraser B., Lake D., Finnegan J., Viljoen N., Miller M. and Rao N. Are we ready for sdn? Implementation challenges for software-defined networks IEEE Communications Magazine, 2013, 51 (7), pp. 36-43.
- [4]. Mohammed, A. H., Khaleefah, R. M., k. Hussein, M., and Amjad Abdulateef, I. A review software defined networking for internet of things. In 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 2020, pp. 1–8.
- [5]. OpenNetworkingFoundation (2012). Software-defined networking: The new norm for networks. ONF White Paper. 2012.
- [6]. Burdonov, I.; Kossachev, A.; Yevtushenko, N.; López, J.; Kushik, N. and Zeghlache, D. (2021). Preventive Model-based Verification and Repairing for SDN Requests. In Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE, ISBN 978-989-758-508-1 ISSN 2184-4895, pp. 421-428. DOI: 10.5220/0010494504210428.
- [7]. Igor Burdonov, Nina Yevtushenko and Alexander Kossatchev. Implementing a virtual network on the SDN data plane. Proceedings 2020 IEEE East-West Design & Test Symposium (EWDTS). 2020, pp. 279- 283.
- [8]. Бурдонов И.Б., Евтушенко Н.В., Косачев А.С. Реализация распределенных и параллельных вычислений в сети SDN. Труды института системного программирования. 2022, т. 34, № 3, с. 159- 172.
- [9]. Оре О. Теория графов. М.: Наука, 1968, с. 59, 68.

ПРИЛОЖЕНИЕ

Типы глобальных переменных и параметров процедур записаны *строчными буквами полужирным курсивом*.

Имена процедур *Начинаются с прописной буквы и записаны полужирным курсивом*.

Имена глобальных переменных *Начинаются с прописной буквы и записаны не полужирным курсивом*.

Имена параметров процедур и локальных переменных *начинаются со строчной буквы и записаны не полужирным курсивом*.

П1. Типы глобальных переменных и параметров процедур в хостах и коммутаторах

bool – булевский тип,

vertex – идентификатор вершины графа,

service – имя услуги,

set(type) – множество элементов типа *type*,

list(type) – список элементов типа *type*,

(type₁, type₂) – пара (элемент типа *type₁*, элемент типа *type₂*).

П2. Глобальные переменные в вершине – хосте или коммутаторе

vertex Self = ...; /* инициализировано, не меняется: собственный идентификатор вершины */

list(vertex) Neighbors = ...; /* инициализировано, не меняется: список соседних вершин */

bool Repeat = **false**; /* инициализировано первоначально как false; */

/* во время настройки получает значение **true**, когда вершина получает первое сообщение */

list(vertex, vertex) Rules; /* список правил коммутации как список пар (предшественник, преемник) */

bool Host = ...; /* инициализировано, не меняется: признак того, что вершина является хостом */

П3. Глобальные переменные в хосте

set(service) Services = ...; /* инициализировано: множество имён услуг, реализуемых хостом */

Bool Root; /* признак того, что хост является корнем дерева хостов */

П4. Использование глобальных переменных

Глобальные переменные, используемые на этапе самонастройки сети: *Neighbors*, *Repeat*, *Rules*, *Host*, *Root* (все, кроме *Self*, *Services*).

Глобальные переменные, используемые при передаче сообщений по настроенной сети: *Self*, *Rules*, *Host*, *Services*, *Root* (все, кроме *Neighbors*, *Repeat*); при генерации сообщений: *Self*, *Rules*, *Root*.

П5. Правила коммутации

Правила коммутации *Rules* представлены как список длиной *n* пар (идентификатор соседа-предшественника, идентификатор соседа-преемника): (a_0, a_1) , (a_1, a_2) , ..., (a_{n-1}, a_n) . Мы будем использовать следующую обычную нотацию:

$Rules[i..j] = (a_{i-1}, a_i), \dots, (a_{j-1}, a_j)$, если $i = 1..n$, $j = 1..n$ и $i \leq j$; иначе = ();

$Rules[i][1] = a_{i-1}$ для $i = 1..n$;

$Rules[i][2] = a_i$ для $i = 1..n$.

Правило умолчания: при получении вершиной сообщения от вершины *b*, отличной от вершин $a_0, \dots, a_{n-1}$, оно пересылается родителю a_0 (предшественнику из первого правила) по подразумеваемому правилу (b, a_0) .

П6. Вспомогательные процедуры

```
vertex NextNeighbor(vertex x) { /* следующий после x сосед в циклическом списке  $b_1, \dots, b_k$  */
|   n = |Neighbors|; i := 1;
|   while i ≤ n & Neighbors[i] ≠ x do { i := i + 1; }
|   if i = n { i := 0; } /* если  $x = b_i$  и  $i < n$ , то  $b_{i+1}$ , если  $x = b_k$ , то  $b_1$  */
|   return Neighbors[i + 1]; }
```

```
vertex Successor(list(vertex, vertex) rules, vertex x) {
/* по предшественнику x вычисление преемника b в списке правил  $rules = (a_0, a_1), \dots, (a_{n-1}, a_n)$  */
|   n = |rules|; i := 1;
|   while i ≤ n & rules[i][1] ≠ x do { i := i + 1; }
|   if i ≤ n { return rules[i][2]; }
|   else return rules[1][1]; /* если для каждого b в rules нет правила (x, b), возвращаем  $a_0$  */
```

П7. Процедуры обработки сообщений

Сигнатура процедуры обработки сообщения имеет вид **Type**(vertex x, параметры сообщения), а оператор посылки сообщения имеет вид **SEND**(Type(параметры сообщения), y), где **Type** тип сообщения, x сосед, от которого принято сообщение, y сосед, которому посылается сообщение.

П7.1 Самонастройка сети

Построение цикла хостов (цикла обхода дерева хостов с корнем в инициаторе) по алгоритму Тэрри, а также леса деревьев коммутаторов.

vertex x – идентификатор соседа, от которого получено сообщение,

```
Start(vertex x) { /* старт настройки, сообщение пришло в хост-инициатор от его внешнего соседа x */
|   Rules := (); /* удаляем все правила коммутации в вершине */
|   Root := true; /* инициатор становится корнем дерева хостов */
|   if Neighbors = () { /* если корень изолированная вершина (нет соседей), то */
|   |   SEND(Back(), x); /* ответ внешнему соседу, конец настройки */
|   |   else { /* у корня есть соседи */
|   |   |   Repeat := true; /* отмечаем, что корень принял первое сообщение настройки */
|   |   |   Rules := ((x, NextNeighbor(x))); /* создаём правило (внешний сосед,  $a_1$ ),  $a_1$  = первый сосед */
|   |   |   SEND(Forward(), NextNeighbor(x)); /* прямое сообщение соседу  $a_1$  */
```

```
Forward(vertex x) { /* прямое сообщение послано по ещё не пройденной дуге */
|   if Repeat = false { /* если это первое сообщение настройки, принятое вершиной, то */
|   |   Repeat := true; /* отмечаем, что вершина приняла первое сообщение настройки */
|   |   Rules := ((x, NextNeighbor(x))); /* создаём правило (x,  $a_1$ ) от родителя  $a_0 = x$  к соседу  $a_1$  */
|   |   if Host = true { Root := false; } /* если это хост, то отмечаем, что хост не корень */
|   |   if NextNeighbor(x) ≠ x { /* если вершина не листовая в остоном дереве,  $a_1 \neq a_0 = x$ , то */
|   |   |   SEND(Forward(), NextNeighbor(x)); /* прямое сообщение следующему соседу  $a_1$  */
|   |   |   else { /* если вершина листовая в остоном дереве,  $a_1 = a_0 = x$ , то */
|   |   |   |   Repeat := false; /* очистка для следующей настройки */
|   |   |   |   if Host = true { /* если хост, то */
|   |   |   |   |   SEND(Back(), x); /* положительный ответ родителю  $a_0 = x$  */
|   |   |   |   |   else { /* если не хост, то «Удаление «лишних» коммутаторов» */
|   |   |   |   |   |   SEND(Cancel(), x); } } /* отрицательный ответ родителю  $a_0 = x$  */
|   |   |   else { /* повторное прямое сообщение, пришло по хорде остоного дерева */
|   |   |   |   SEND(Cancel(), x); } } /* отрицательный ответ соседу x */
```

```

Cancel(vertex  $x$ ) { /* отрицательный ответ от вершины к её родителю или по хорде остовного дерева */
    /* в этот момент правила коммутации  $Rules = (a_0, a_1), (a_1, a_2), \dots, (a_{n-2}, a_{n-1}), (a_{n-1}, a_n = x)$  */
     $n := |Rules|$ ; /* число правил коммутации */
     $y := NextNeighbor(x)$  /* сосед  $y$  следующий после соседа  $x$  в циклическом списке соседей */
     $Rules[n][2] := y$ ; /* меняем последнее правило  $(a_{n-1}, a_n = x) \rightarrow (a_{n-1}, y)$  */
    if  $y \neq Rules[1][1]$  /* если  $y \neq a_0$  (не родитель), то не все соседи, отличные от родителя, пройдены */
    |   SEND(Forward(),  $y$ ); /* прямое сообщение не пройденному соседу  $y$  */
    else { /* если все соседи, отличные от родителя, пройдены, то */
    |   Repeat := false; /* очистка для следующей настройки */
    |   if  $Host = \text{true}$  or  $n > 1$  /* если вершина хост или не листовая в дереве хостов, то */
    |   |   SEND(Back(),  $y$ ); /* положительный ответ родителю  $y = a_0$  */
    |   |   else { /* если вершина листовой коммутатор, то «Удаление «лишних» коммутаторов» */
    |   |   |   SEND(Cancel(),  $y$ ); /* отрицательный ответ родителю  $y = a_0$  */
    |   |   |   if  $Host = \text{true} \ \& \ Root = \text{true}$  { /* если обход завершён, то конец настройки */
    |   |   |   |   /* замыкание цикла хостов: (внешний,  $a_1$ ), ...,  $(a_{n-1}, \text{внешний}) \rightarrow (a_{n-1}, a_1)$ , ...,  $(a_{n-2}, a_{n-1})$  */
    |   |   |   |    $Rules[1][1] := Rules[n][1]$ ; /* первое правило: (внешний,  $a_1$ )  $\rightarrow (a_{n-1}, a_1)$  */
    |   |   |   |    $Rules := Rules[1..n - 1]$ ; } } /* удаляем последнее правило  $(a_{n-1}, \text{внешний})$  */
    |   |   }
    |   }

Back(vertex  $x$ ) { /* положительный ответ от вершины к её родителю в дереве хостов */
    /* в этот момент правила коммутации  $Rules = (a_0, a_1), (a_1, a_2), \dots, (a_{n-2}, a_{n-1}), (a_{n-1}, a_n = x)$  */
     $y := NextNeighbor(x)$  /* сосед  $y$  следующий после соседа  $x$  в циклическом списке соседей */
     $Rules := Rules \wedge (x, y)$ ; /* добавляем правило  $(a_n = x, a_{n+1} = y)$  */
    if  $y \neq Rules[1][1]$  /* если  $y \neq a_0$  (не родитель), то не все соседи, отличные от родителя, пройдены */
    |   SEND(Forward(),  $y$ ); /* прямое сообщение не пройденному соседу  $a_{n+1} = y$  */
    else { /* если все соседи, отличные от родителя, пройдены, то */
    |   Repeat := false; /* очистка для следующей настройки */
    |   SEND(Back(),  $y$ ); /* положительный ответ родителю  $y = a_0$  */
    |   if  $Host = \text{true} \ \& \ Root = \text{true}$  { /* если обход завершён, то конец настройки */
    |   |    $n := |Rules|$ ; /* число правил коммутации */
    |   |   /* замыкание цикла хостов: (внешний,  $a_1$ ), ...,  $(a_{n-1}, \text{внешний}) \rightarrow (a_{n-1}, a_1)$ , ...,  $(a_{n-2}, a_{n-1})$  */
    |   |    $Rules[1][1] := Rules[n][1]$ ; /* первое правило: (внешний,  $a_1$ )  $\rightarrow (a_{n-1}, a_1)$  */
    |   |    $Rules := Rules[1..n - 1]$ ; } } /* удаляем последнее правило  $(a_{n-1}, \text{внешний})$  */
    |   }

```

П7.2 Передача сообщений по настроенной сети

vertex x – идентификатор соседа, от которого получено сообщение,

vertex sender – идентификатор хоста-отправителя, применяется для посылки ответа отправителю,

service $service$ – имя запрашиваемой услуги,

parameters – параметры сообщения, прозрачные для коммутации сообщений.

```

MessageToHost(vertex  $x$ , vertex sender, vertex recipient, parameters) {
    /* сообщение хосту-получателю, не прошедшее дугу от родителя корня в корень */
     $y := Successor(x)$ ; /* сосед-преемник  $y$  соседа-предшественника  $x$  */
    if  $Self \neq recipient$  { /* если вершина не получатель, то */
    |   if  $x \neq Rules[1][1] \vee Host \neq \text{true} \vee Root = \text{false}$  { /* если пришли не в корень от его родителя, то */
    |   |   SEND(MessageToHost(sender, recipient, parameters),  $y$ ); /* дальше по циклу хостов */
    |   |   else { /* если пришли в корень от его родителя, то */
    |   |   |   SEND(RootMessageToHost(sender, recipient, parameters),  $y$ ); } /* дальше по циклу хостов */
    |   |   else { /* если вершина получатель, то */
    |   |   |   MessageToHostProcessing(parameters); } } /* локальная обработка принятого сообщения */
    |   }

```

RootMessageToHost(vertex x, vertex sender, vertex recipient, parameters) {

```
    /* сообщение хосту-получателю, прошедшее дугу от родителя корня в корень */
    y := Successor(Rules, x); /* сосед-преемник у соседа-предшественника x */
    if Self ≠ recipient { /* если вершина не получатель, то */
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(RootMessageToHost(sender, recipient, parameters), y); /* дальше по циклу хостов */
        } else { /* если пришли по дуге от родителя корня в корень, то */
            /* отрицательный ответ отправителю сообщения, не нашедшего получателя */
            SEND(MessageToHost(Self, sender, parameters), y); } }
    else { /* если вершина хост recipient, то */
        MessageToHostProcessing(parameters); } } /* локальная обработка принятого сообщения */
```

Message(vertex x, vertex sender, service service, parameters) {

```
    /* сообщение запроса услуги, не прошедшее дугу от родителя корня в корень */
    y := Successor(Rules, x); /* сосед-преемник у соседа-предшественника x */
    if Host ≠ true ∨ service ∉ Services { /* если в вершине не реализована запрашиваемая услуга, то */
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(Message(sender, service, parameters), y); /* дальше по циклу хостов */
        } else { /* если пришли по дуге от родителя корня в корень, то */
            SEND(RootMessage(sender, service, parameters), y); } } /* дальше по циклу хостов */
    else { /* если вершина хост, реализующий запрашиваемую услугу, то */
        if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
            SEND(WaitingMessage(sender, service, parameters), y); /* дальше по циклу хостов */
        } else { /* если хост «свободен», то */
            service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */
```

RootMessage(vertex x, vertex sender, service service, parameters) {

```
    /* сообщение запроса услуги, прошедшее дугу от родителя корня в корень */
    y := Successor(Rules, x); /* сосед-преемник у соседа-предшественника x */
    if Host ≠ true ∨ service ∉ Services { /* если в вершине не реализована запрашиваемая услуга, то */
        if x ≠ Rules[1][1] ∨ Host ≠ true ∨ Root = false {
            /* если пришли не в корень или в корень, но не от его родителя, то */
            SEND(RootMessage(sender, service, parameters), y); /* дальше по циклу хостов */
        } else { /* если пришли по дуге от родителя корня в корень, то */
            /* отрицательный ответ отправителю сообщения, не нашедшего получателя */
            SEND(MessageToHost(Self, sender, parameters), Rules[1][2]); } }
    else { /* если вершина хост, реализующий запрашиваемую услугу, то */
        if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
            SEND(WaitingMessage(sender, service, parameters), y); /* дальше по циклу хостов */
        } else { /* если хост «свободен», то */
            service(sender, parameters); } } } /* локальный вызов запрашиваемой услуги */
```

```

WaitingMessage(vertex  $x$ , vertex sender, service service, parameters) {
|   /* сообщение запроса услуги, ожидающее освобождения хоста, который может оказать услугу */
|    $y := \text{Successor}(\text{Rules}, x)$ ; /* сосед-преемник  $y$  соседа-предшественника  $x$  */
|   if  $\text{Host} \neq \text{true} \vee \text{service} \notin \text{Services}$  { /* если в вершине не реализована запрашиваемая услуга, то */
|   |   if  $x \neq \text{Rules}[1][1] \vee \text{Host} \neq \text{true} \vee \text{Root} = \text{false}$  {
|   |   |   /* если пришли не в корень или в корень, но не от его родителя, то */
|   |   |   SEND(WaitingMessage(sender, service, parameters),  $y$ ); /* дальше по циклу хостов */
|   |   |   else { /* если пришли по дуге от родителя корня в корень, то */
|   |   |   |   SEND(Message(sender, service, parameters),  $y$ ); } /* дальше по циклу хостов */
|   |   else { /* если вершина хост, реализующий запрашиваемую услугу, то */
|   |   |   if HostReadyToExecuteService(service, parameters) = false { /* если хост «занят», то */
|   |   |   |   SEND(WaitingMessage(sender, service, parameters),  $y$ ); /* дальше по циклу хостов */
|   |   |   |   else { /* если хост «свободен», то */
|   |   |   |   |   service(sender, parameters); } } /* локальный вызов запрашиваемой услуги */
|   |   }
|   }
}

```

П8. Генерация сообщения хосту и вызов удалённой услуги

vertex *recipient* – идентификатор хоста, которому предназначено сообщение.

service *service* – имя запрашиваемой услуги,

parameters – параметры сообщения, прозрачные для коммутации сообщений,

```

bool SendMessageToHost(vertex recipient, parameters); {
|   /* генерация сообщения известному хосту-получателю */
|   if  $\text{Rules} = ()$  { /* если нет правил, то */
|   |   return false; }
|   if  $\text{Root} = \text{false}$  { /* если вершина не корень, то */
|   |   /* посылаем сообщение по циклу хостов преемнику  $a_1$  первого правила ( $a_0, a_1$ ) */
|   |   SEND(MessageToHost(Self, recipient, parameters),  $\text{Rules}[1][2]$ ); }
|   else { /* если вершина корень, то */
|   |   /* посылаем сообщение по циклу хостов преемнику  $a_1$  первого правила ( $a_0, a_1$ ) */
|   |   SEND(RootMessageToHost(Self, recipient, parameters),  $\text{Rules}[1][2]$ ); }
|   return true; }

```

```

bool SendMessage(service service, parameters); { /* вызов из хоста удалённой услуги */
|   if  $\text{Rules} = ()$  { /* если нет правил, то */
|   |   return false; }
|    $y := \text{Rules}[1][2]$ ; /* посылаем следующему соседу  $a_1$  */
|   if  $\text{service} \notin \text{Services}$  { /* если в хосте не реализована запрашиваемая услуга, то */
|   |   if  $\text{Root} = \text{true}$  { /* если хост корень, то */
|   |   |   SEND(RootMessage(Self, service, parameters),  $y$ ); /* по циклу хостов */
|   |   |   else { /* если хост не корень, то */
|   |   |   |   SEND(Message(Self, service, parameters),  $y$ ); } /* по циклу хостов */
|   |   else { /* если в хосте реализована запрашиваемая услуга (но хост сейчас занят), то */
|   |   |   SEND(WaitingMessage(Self, service, parameters),  $y$ ); /* по циклу хостов */
|   |   return true; }
|   }
}

```

Информация об авторах / Information about authors

Игорь Борисович БУРДОНОВ – доктор физико-математических наук, главный научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, технология компиляции, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Igor Borisovich BURDONOV – Dr. Sci. (Phys.-Math.), a Leading Researcher of ISP RAS. Research interests: formal specifications, test generation, compilation technology, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

Нина Владимировна ЕВТУШЕНКО, доктор технических наук, профессор, главный научный сотрудник ИСП РАН, до 1991 года работала научным сотрудником в Сибирском физико-техническом институте. С 1991 г. работала в ТГУ профессором, зав. кафедрой, зав. лабораторией по компьютерным наукам. Её исследовательские интересы включают формальные методы, теорию автоматов, распределённые системы, протоколы и тестирование программного обеспечения.

Nina Vladimirovna YEVTUSHENKO, Dr. Sci. (Tech.), Professor, a Leading Researcher of ISP RAS, worked at the Siberian Scientific Institute of Physics and Technology as a researcher up to 1991. In 1991, she joined Tomsk State University as a professor and then worked as the chair head and the head of Computer Science laboratory. Her research interests include formal methods, automata theory, distributed systems, protocol and software testing.

Александр Сергеевич КОСАЧЕВ – кандидат физико-математических наук, ведущий научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, технология компиляции, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Alexander Sergeevitch KOSSATCHEV – Cand. Sci. (Phys.-Math.), a Leading Researcher of ISP RAS. Research interests: formal specifications, test generation, compilation technology, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.

Вера Николаевна ПОНОМАРЕНКО – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, генерация тестов, системы реального времени, операционные системы, объектно-ориентированное программирование, сетевые протоколы, процессы разработки программного обеспечения.

Vera Nikolaevna PONOMARENKO – Cand. Sci. (Phys.-Math.), a Senior Researcher of ISP RAS. Research interests: formal specifications, test generation, real-time systems, operating systems, object-oriented programming, network protocols, software development processes.



Алгоритм оптимизации «черепаха и заяц» с адаптивной стратегией на основе взаимной информации для обнаружения сетевых вторжений

Т. Бхуванешвари, ORCID: 0000-0002-4403-307X <bhuvaneswarit@mepcoeng.ac.in>

К. Руба Соундар, ORCID: 0000-0003-1300-6519 <rubasoundar@mepcoeng.ac.in>

Р. Чандра Гуру Секар, ORCID: 0000-0002-1436-1785 <chandraturusekar@mepcoeng.ac.in>

*Технический колледж Мепко Шленк,
Индия, 626005, Тамилнад, Сивакаси, Вирудхунагар.*

Аннотация. В современную эпоху сильной взаимосвязанности данные и информация постоянно передаются по сетям. Обеспечение безопасности конфиденциальной информации и защиты компьютерных систем от сетевых угроз стало очень актуально. Поэтому важна разработка эффективной системы обнаружения вторжений в сеть (NIDS) с использованием оптимальных признаков. Эти признаки могут быть определены с помощью искусственного интеллекта путем изучения шаблонов и взаимосвязей методами машинного обучения (machine learning, ML). В статье представлена методика оптимизации типа «черепаха и заяц» для выбора оптимальных признаков. Для оценки используется набор данных UNSW-NB15. Результаты оптимизации достигают точности 94,12% для бинарной классификации и 93,92% для многоклассовой классификации, при этом из всего набора признаков выбираются 26 оптимальных. Чтобы улучшить подход, используется адаптивная стратегия на основе взаимной информации для управления количеством оптимальных признаков. Эта стратегия вместе с алгоритмом черепахи и зайца повышает точность, показывая 94,69% для бинарной классификации и 94,03% для многоклассовой классификации, при этом сокращая количество выбранных признаков до 9. Сравнительный анализ производительности показывает, что предлагаемый метод выбора признаков превосходит другие современные методы, обеспечивая более точные и надежные результаты при выявлении киберугроз. Кроме того, график связи количества оптимальных признаков и точности модели показывает, что выбор только 9 признаков является эффективным для достижения высокой точности обнаружения и прогнозирования кибератак.

Ключевые слова: система обнаружения сетевых вторжений (NIDS); алгоритм метаэвристической оптимизации; алгоритм оптимизации «черепаха и заяц» (RTOA); выбор признаков (FS); выбор признаков на основе оболочки; взаимная информация (MI).

Для цитирования: Бхуванешвари Т., Руба С. К., Чандра Г. С. Р. Алгоритм оптимизации «черепаха и заяц» с адаптивной стратегией на основе взаимной информации для обнаружения сетевых вторжений. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 31–50. DOI: 10.15514/ISPRAS–2025–37(4)–2.

Благодарности: Авторы благодарят анонимных рецензентов за их предложения и замечания по улучшению качества статьи.

Rabbit and Tortoise Optimization Algorithm with Mutual Information Based Adaptive Strategy for Network Intrusion Detection

T. Bhuvaneswari, ORCID: 0000-0002-4403-307X <bhuvaneswarit@mepcoeng.ac.in>

K. Ruba Soundar, ORCID: 0000-0003-1300-6519 <rubasoundar@mepcoeng.ac.in>

R. Chandra Guru Sekar, ORCID: 0000-0002-1436-1785 <chandragurusekar@mepcoeng.ac.in>

Mepco Schlenk Engineering College,
Virudhunagar, Sivakasi, Tamil Nadu, 626005, India.

Abstract. In the modern era of highly interconnectedness, data and information are constantly transmitted over networks. Ensuring the security of confidential information and protecting computer systems from network threats has become very important. Therefore, it is important to develop an effective network intrusion detection system (NIDS) using optimal features. These optimal features can be identified through computational intelligence by learning patterns and relationships among features using machine learning techniques. This paper presents a Rabbit and Tortoise optimization technique for selecting optimal features. For evaluation, the UNSW-NB15 dataset is utilized. The optimization results achieve an accuracy of 94.12% for binary classification and 93.92% for multi-class classification, with 26 optimal features selected from the entire feature set. To improve the approach, an adaptive strategy based on mutual information is used to control the number of optimal features. This strategy, together with the Rabbit and Tortoise algorithm, improves the accuracy, showing 94.69% for binary classification and 94.03% for multi-class classification, while reducing the number of selected features to 9 only. The comparative performance analysis shows that the proposed feature selection method outperforms other state-of-the-art methods, providing more accurate and reliable results in identifying cyber threats. In addition, the relationship plot between the number of optimal features and the accuracy of the model shows that selecting only 9 features is effective in achieving high accuracy in detecting and predicting cyber-attacks.

Keywords: network intrusion detection system (NIDS); metaheuristic optimization algorithm; rabbit and tortoise optimization algorithm (RTOA); feature selection (FS); wrapper-based feature selection; mutual information (MI).

For citation: Bhuvaneswari T., Ruba Soundar K., Chandra Guru Sekar R. Rabbit and Tortoise optimization algorithm with mutual information based adaptive strategy for network intrusion detection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 31-50 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-2.

Acknowledgements. The authors thank anonymous reviewers for their suggestions and comments to improve the quality of the article.

1. Введение

В современном взаимосвязанном мире инновации стали жизненно важным компонентом повседневной жизни. Люди в значительной степени полагаются на Интернет, искусственный интеллект, пользуясь такими устройствами, как смартфоны и умные гаджеты. Однако растущая зависимость от онлайн-сетей и устройств чревата растущим риском киберугроз. Эта уязвимость распространяется и на цифровые активы, принадлежащие предприятиям, включая сетевые системы. Со временем частота кибератак только растет, а их номенклатура постоянно расширяется [1]. Такие атаки приобрели международный характер, что привело к заметному снижению общей продуктивности мировой экономики. Чтобы преодолеть эти угрозы безопасности, компьютерные системы должны быть оснащены инструментами для проведения регулярного мониторинга и анализа поведенческих моделей, как для рутинных, так и для необычных действий. Концепция систем обнаружения вторжений (СОВ) играет важную роль в выявлении и устранении новых угроз безопасности и сетевых атак [2].

Функции обнаружения вторжений включают наблюдение и анализ действий пользователей и системы, оценку настроек и уязвимостей систем, оценку целостности систем и файлов, выявление семейств и шаблонов атак, изучение нестандартного поведения и мониторинг

нарушений политик пользователя. Сетевые вторжения обнаруживаются на основе анализа характеристик сетевого трафика.

Большинство современных систем обнаружения сетевых вторжений (network intrusion detection system, NIDS) идентифицируют атаки, анализируя все признаки, полученные из переданных по сети данных. При этом вычислительная стоимость анализа всех атрибутов очень высока. Однако для эффективного обнаружения атак не все признаки необходимы. Более того, показано, например, что избыток признаков в наборе данных UNSW-NB15 [1] приводит к переобучению и увеличению времени обнаружения вторжений.

Применение методов машинного обучения (machine learning, ML) в сочетании с отбором признаков в наборе данных UNSW-NB15 может облегчить идентификацию важных атрибутов для обнаружения вторжений. Уменьшение количества признаков может привести к сокращению времени обнаружения без ущерба для точности. Основная цель отбора признаков – идентификация и отбор признаков по степени их важности. Выбор признаков нашел применение в различных областях исследований, например, в таких, как обнаружение вторжений [3], прогнозирование болезни Паркинсона [4], прогнозирование рака [5] и многих других.

Методы отбора признаков можно разделить на три группы: фильтрация, обертка и гибридный метод. При фильтрации для определения критических признаков используют методы статистики, теории информации и теории множеств. Отбор признаков на основе оберток – это один из методов интеллектуального анализа данных, который оценивает и выбирает подмножества признаков с учетом их производительности в рамках выбранной модели машинного обучения. Процесс включает в себя итеративное обновление подмножеств признаков с помощью стратегии поиска и алгоритма оптимизации. Гибридный выбор признаков объединяет два или более методов выбора признаков, как правило, из методов фильтрации и обертки, чтобы использовать их сильные стороны и смягчить ограничения.

Последние исследования по этой теме используют методы мета-эвристической оптимизации как стратегию поиска для решения самых сложных задач. В этой статье мы представляем алгоритм оптимизации «черепаха и заяц» (Rabbit and Tortoise optimization algorithm, RTOA) и адаптивный алгоритм оптимизации «черепаха и заяц» (адаптивный RTOA, A-RTOA) для отбора признаков. Алгоритм RTOA рассматривается как стратегия поиска в методе обертки. Адаптивный алгоритм A-RTOA – это гибридный метод, который объединяет RTOA на основе обертки и взаимную информацию на основе фильтра в рамках адаптивной стратегии.

Статья организована следующим образом: в разделе 2 содержится обзор современных методов обнаружения вторжений и методов выбора признаков. В разделе 3 представлены и математически определены алгоритмы RTOA и A-RTOA. В разделе 4 обсуждается набор данных для NIDS и реализация предлагаемых для выбора признаков алгоритмов RTOA и A-RTOA. В разделе 5 экспериментальные результаты сравниваются с аналогичными методами в форме таблиц/столбчатых диаграмм, показывается и обсуждается важность адаптивного метода. В Заключение представлены основные выводы, намечена будущая работа.

2. Обзор литературы

Тахир и др. предложили восстанавливать пропущенные данные на основе глубокого обучения для улучшения качества данных в системах обнаружения вторжений (intrusion detection system, IDS). Метод использует технику Random Missing Value для имитации пропущенных данных, что позволяет проводить тщательную оценку различных вычислительных подходов. Эксперименты проводились на наборах данных NSL-KDD и UNSW-NB15 [6]. Сахид и др. представили метод ансамблевого обучения на основе обнаружения атакующих вторжений в сетях Интернета вещей (IoT) с помощью оптимизатора Gray Wolf Optimizer на наборах данных BoT IoT и UNSW-NB15 [7]. Чжу и др. интегрировали метод ансамблевого обучения и кластеризации подпространства для обнаружения вторжений

в сетях интернета вещей [8]. Халладжи и др. предложили структуру для классификации по нескольким меткам в COB, которая преодолевает ограничения традиционных методов с одной меткой [9].

Ли и др. представили SELSTM (Selective Encoding LSTM) – усовершенствованную систему IDS для Интернета вещей, основанную на сети сжатия и возбуждения [10] и объединяющую концепции нейронной сети семантического встраивания (neural semantic embedding network, NSENet) и долгой краткосрочной памяти (long short-term memory, LSTM). Диша и др. представили систему обнаружения вторжений, которая использует модели машинного обучения для защиты сетей и данных. Взвешенный случайный лес (Random Forest, RF) с примесью Джини используется для обработки многомерных данных для выбора признаков в наборе данных UNSW-NB15 [11]. Кесервани и др. представили всесторонний обзор систем COB, выделив их типы: основанные на неправильном использовании, основанные на аномалиях и гибридные, и обсудив проблемы, возникающие из-за увеличения сложности сети и объема данных. Рассматриваются шесть эталонных наборов данных и различные методы снижения размерности и классификации (в частности, машинное и глубокое обучение), способствующие повышению эффективности и надежности COB [12]. Юсефнежад и др. представили систему COB, использующую ансамблевые модели. Система использует опорные векторные машины (support vector machines, SVM) для обнаружения нормального трафика и метод k-ближайших соседей (k-nearest neighbors, kNN) для многоклассовой классификации [13].

Кабилан и др. предложили неконтролируемый метод обнаружения вторжений для сетей связи в автомобиле, использующий автокодировщики для извлечения признаков и нечеткую кластеризацию C-средних для точного обнаружения [14]. Алаззам и др. предложили оптимизацию процедуры выбора признаков, “вдохновленную голубем” (pigeon-inspired optimization) [15]. Фероз Хан и Анандхарадж представили многоуровневый подход к анализу проблем безопасности на основе пороговых контрмер против атак повторного воспроизведения на каждом уровне [16]. Для идентификации вторжений Мегантара и Тохари интегрировали отбор признаков с использованием контролируемого обучения и сокращение данных посредством неконтролируемого обучения на наборе данных NSL-KDD [17].

Альмиани и др. показали, как использовать глубокие рекуррентные нейронные сети (deep recurrent neural networks) и контролируемые методы управления устройствами для классификации и прогнозирования внезапных скоординированных атак для масштабирования устойчивых систем COB для медицинского интернета вещей. Затем была выполнена оптимизация функций для обнаружения вторжений с использованием “биологически вдохновленных” алгоритмов роя частиц [18].

Бхаттачарья и др. опубликовали гибридный анализ главных компонент (principal component analysis, PCA) и метод машинного обучения, используемый в качестве основы для классификации записей IDS. В предлагаемом ими алгоритме проводится быстрое кодирование массивов данных, находящихся в распоряжении систем COB, а уменьшение размерности выполняется на основе гибридной модели “мотылька” (firefly model) PCA [19]. Moualla et al. использовали критерий примесей Джини для отбора важных признаков [20]. Сахид и Ароволо исследовали проблемы повышения безопасности медицинского интернета вещей с помощью контролируемых моделей машинного обучения, таких как случайный лес и дерево решений, а также глубокие рекуррентные нейронные сети [21].

Девпраasad и др. предложили систему обнаружения вторжений на основе аномалий. Они провели эксперименты на наборах данных NSL-KDD и UNSW-NB15, в которых использовали методику определения порядка предпочтения по сходству с идеальным решением, механизм ранжирования, включающий хи-квадрат и алгоритм летучей мыши для выбора признаков [22]. Ширавани и др. представили новый подход к выбору признаков с использованием оценки на основе корреляции и нечетких чисел (fuzzy numbers) для систем обнаружения вторжений. Определяя признаки как нечеткие числа и оптимизируя с помощью

эвристической функции, он эффективно уменьшает размер данных. Экспериментальные результаты на наборах данных KDD Cup, NSL-KDD и CICIDS демонстрируют более высокие показатели обнаружения и эффективность по сравнению с традиционными методами, такими как выбор признаков на основе корреляции [23].

Касонголо и др. проанализировали набор данных обнаружения вторжений UNSW-NB15 для обучения систем COB. Применяется фильтрующее сокращение признаков с использованием библиотеки XGBoost (extreme gradient boosting) [24]. Kumar et al. представили новую систему COB, разработанную на основе неправильного использования (misuse) для классификации сетевых угроз. Они применили набор данных UNSW-NB15 для обучения и оценки модели с новым набором данных в реальном времени RTNITP18, который был создан в NIT Patna [25]. Мефтах С. И др.. использовали двухэтапный процесс обнаружения аномалий с набором данных UNSW-NB15 и продемонстрировали оценку эффективности модели как в бинарной классификации, так и в мультиклассификации [26]. Альмомани представил метод выбора признаков с использованием

1. генетических алгоритмов;
2. оптимизации, вдохновлённой мигающим поведением светлячков (firefly);
3. оптимизатора Grey Wolf и
4. оптимизации роя частиц (Particle swarm optimization) для выбора атрибутов в наборе данных UNSW-NB15 с использованием классификаторов J48 и опорных векторных машин [27].

Ахмад и др. исследовали проблемы и достижения в области сетевых COB. Они рассмотрели методы машинного и глубокого обучения для улучшения производительности COB [28].

Белух и др. представили двухступенчатый классификатор с использованием алгоритма RepTree (reduced error pruning tree) и подмножеств протоколов, оцененных с наборами данных UNSW-NB15 и NSL-KDD для обнаружения сетевых вторжений [29]. Несколько авторов внесли вклад в методы выбора признаков в справочных статьях [30-39].

Описанные работы подчеркивают важность методов выбора признаков для повышения производительности классификатора. Однако адаптивным метаэвристическим подходам, нацеленным на обнаружение вторжений, в существующих исследованиях уделено мало внимания. В этой статье предлагается новый метаэвристический алгоритм оптимизации RTOA, интегрирующий с адаптивными стратегиями (A-RTOA) для устранения этих пробелов, с упором на улучшенный отбор признаков для NIDS. Как часть реализации предлагаемого алгоритма авторы использовали набор данных UNSW-NB15, доступность данных подтверждена в [40]. Алгоритмы RTOA и A-RTOA использовались в процессе отбора признаков для обнаружения вторжений в сеть. Оптимальные признаки выбирались и анализировались с использованием пяти классификаторов: случайного леса [41], CatBoost [42], XGBoost [43], KNN [44] и дерева решений [45].

3. Предлагаемый метод

Этот раздел посвящен описанию алгоритмов RTOA и A-RTOA, предназначенных для отбора признаков. Демонстрируется важность адаптивной стратегии.

3.1 Алгоритм оптимизации «черепаха и заяц»

Как обсуждалось во введении, алгоритм оптимизации «черепаха и заяц» (RTOA) – это подражающий природе метаэвристический алгоритм оптимизации. Он вдохновлен известной историей о зайце и черепахе и имитирует характеристики этих животных: высокую скорость и непоследовательность зайца и медленное, но устойчивое движение черепахи. Эти характеристики используются для эффективного поиска оптимальных решений в задачах оптимизации (в основном в процессах выбора признаков).

Алгоритм RTOA моделирует поведение зайца и черепахи:

1. Быстрые движения зайца (высокая скорость).
2. Размеренное и неспешное движение черепахи (низкая скорость).

Объединяя эти две различные характеристики, RTOA эффективно перемещается по ландшафту оптимизации, балансируя между исследованием нового и эксплуатацией возможностей известного для достижения оптимальных решений. В литературе, помимо прочего, алгоритм используется для обнаружения цикла в связанном списке [46]. Однако предлагаемый в этой статье вариант RTOA специально разработан для обеспечения надежного и эффективного подхода к решению сложных задач оптимизации, особенно в контексте отбора признаков. В алгоритме шаги с 1 по 8, за исключением шага 5, представляют собой процесс RTOA.

3.2 Вклад и структура

RTOA имеет ряд существенных особенностей.

1. Это метаэвристический метод оптимизации, который снижает необходимость в обширной настройке параметров за счет использования только двух параметров, обозначаемых как α и β .
2. RTOA вдохновлен контрастным поведением зайца и черепахи. Он имитирует быстрый, но непоследовательный темп зайца и медленный, но уверенный ход черепахи.
3. Алгоритм использует динамику зайца и черепахи для управления процессом оптимизации, сочетая быстрое исследование с устойчивой эксплуатацией для поиска оптимальных решений.
4. RTOA используется для выбора соответствующих подмножеств признаков при обнаружении вторжений (UNSW-NB15), тем самым повышая производительность систем обнаружения за счет эффективного выбора признаков.
5. Предложенный метод RTOA был тщательно протестирован на известном наборе данных (UNSW-NB15) с использованием пяти популярных алгоритмов классификации (CatBoost, XGBoost, KNN, дерево решений и случайный лес), продемонстрировав свою эффективность и надежность в реальных сценариях.
6. Через контроль над количеством признаков предлагается адаптивная стратегия на основе взаимной информации с алгоритмом оптимизации «черепаха и заяц», которая обозначается как A-RTOA. Результаты A-RTOA сравниваются с другими методами, а также с обычным RTOA.

3.3 Адаптивная оптимизация зайца и черепахи на основе взаимной информации (A-RTOA)

Значение адаптивной стратегии заключается в преодолении ограничений неадаптивных подходов в выборе признаков. В неадаптивных стратегиях иногда возможно, что к концу максимальной итерации не будет выбрано ни одного признака (то есть двоичный вектор имеет значение (0, 0, 0, 0, ...) в последней итерации). В этой ситуации он не только теряет время, но и не может дать эффективного решения. Адаптивная стратегия помогает обеспечить успех в выборе определенного числа оптимальных подмножеств признаков.

В области методов оптимизации адаптивная метаэвристика представляет собой метод, который использует самонастраивающиеся стратегии метаэвристической оптимизации для выбора подмножеств признаков. В подходе на основе обертки для выравнивания количества признаков в каждой итерации используются адаптивные стратегии, которые для конкретной задачи выбирают наиболее оптимальные признаки с помощью итерационного подхода.

Ключевым преимуществом адаптивной стратегии является ее способность обеспечивать контроль над количеством выбранных признаков при сохранении значения пригодности. В работе используется основанная на фильтре взаимная информация (ВИ) [41] для балансировки количества признаков между итерациями в RTOA. Этот метод особенно ценен для повышения эффективности моделей NIDS и снижения сложности набора данных за счет концентрации на наиболее информативных признаках. Такой подход является новым по сравнению с существующими исследованиями. В настоящем алгоритме A-RTOA представлен шагами от 1 до 8 (включая шаг 5).

3.4 Формулировка RTOA и A-RTOA

RTOA разработан в соответствии с вышеупомянутыми характеристиками зайца и черепахи. Предположим, что задача включает в себя D переменных, а популяция состоит из n_c выбранных решений-кандидатов. В нашей работе D представляет собой общее количество признаков в рассматриваемом наборе данных. Каждый кандидат представлен так:

$$v_k = (v_{k1}, v_{k2}, v_{k3}, \dots, v_{kD})$$

где k меняется от 1 до n_c .

3.5 Алгоритм RTOA и A-RTOA

Шаг 1: Инициализация

Число кандидатов в признаки (n_c) определено. Во время инициализации каждый кандидат случайным образом выбирает признаки следующим образом:

$$v_{kj}^0 = \text{Случайный}\{0,1\}$$

где:

- v_{kj}^0 – начальное значение (в момент времени 0) для j -того признака ($j = 1, 2, 3, \dots, D$) k -того кандидата ($k = 1, 2, 3, \dots, n_c$),
- Случайный $\{0, 1\}$ означает случайное назначение 0 или 1.
- Если, $v_{kj}^0 = 1$, j -тый признак набора данных учитывается для процесса оценки, в противном случае – нет.

Шаг 2: Первоначальный расчет приспособленности и выбор глобально лучшего кандидата

Начальная пригодность каждого кандидата $v_k^0 = (v_{k1}^0, v_{k2}^0, v_{k3}^0, \dots, v_{kD}^0)$ оценивается с помощью классификатора на основе выбранных им признаков v_{ki}^0 . Кандидат с максимальной пригодностью определяет v_{best}^0 двоичный вектор, представляющий начальное наилучшее решение.

Шаг 3: Обновление позиций кандидатов

Как описано выше, алгоритм обновления позиций Зайца и Черепахи каждого кандидата v_k^{t+1} состоит из двух фаз: «Фаза Зайца» и «Фаза Черепахи» из предыдущей позиции v_k^t . Фазы определяются на основе случайности. Если Случайный $\{0,1\} < 0.5$, то будет выполнена Фаза Зайца, в противном случае будет выполнена Фаза Черепахи. Подробное описание выглядит следующим образом:

Шаг 3-1: Фаза Зайца. На этом этапе предпринимаются более масштабные шаги для исследования новых областей пространства решений (то есть параметра скорости движения $0.5 \leq \alpha \leq 1$).

$$\begin{aligned}\text{Шаг_зайца} &= \alpha * |v_{\text{лучший},j}^t - v_{k,j}^t| \\ v_{k,j}^{t+1} &= v_{k,j}^t + \text{Шаг_зайца}\end{aligned}$$

Шаг 3-2: Фаза черепахи. На этом этапе делаются более мелкие и точные шаги, использующие известные хорошие решения (то есть параметр скорости движения $0 < \beta < 0.5$)

$$\begin{aligned}\text{Шаг_черепахи} &= \beta * |v_{\text{лучший},j}^t - v_{k,j}^t| \\ v_{k,j}^{t+1} &= v_{k,j}^t + \text{Шаг_черепахи}\end{aligned}$$

где,

- $v_{k,j}^t$ является текущим решением k -того отдельного признака в группе.
- $v_{\text{лучший},j}^t$ является наилучшим решением, которое в настоящее время было найдено.
- α — коэффициент, контролирующий размер шага исследования.
- β — коэффициент, контролирующий размер шага эксплуатации.
- Термин $|v_{\text{лучший},j}^t - v_{k,j}^t|$ представляет собой движение к лучшему кандидату.

Шаг 4: Обновление каждого кандидата для следующей итерации

Векторы решения v_k^{t+1} шага 3 могут не быть бинарными векторами. Следующее правило обновляет j -тую позицию k -того кандидата в итерации $t+1$, превращая v_k^{t+1} в бинарный вектор.

$$v_{k,j}^{t+1} = \begin{cases} 1, & P_{k,j}^t > \text{Случайный}(0,1) \\ 0, & \text{Otherwise} \end{cases}$$

Пространство поиска $P_{k,j}^t$ определяется как,

$$P_{k,j}^t = \frac{(e^{v_{k,j}^t} - e^{-v_{k,j}^t})}{(e^{v_{k,j}^t} + e^{-v_{k,j}^t})}$$

где,

- $v_{k,j}^{t+1}, v_{k,j}^t$ — обозначают текущее и предыдущее значение j -той позиции в k -том кандидате.
- Случайный(0,1) обозначает случайное число, которое имеет равномерное распределение в диапазоне от 0 до 1.

Шаг 5: Выбор признаков с использованием концепции адаптивной стратегии на основе взаимной информации

В алгоритме RTOA шаг 5 пропускается, и процесс продолжается переходом к шагу 6. В алгоритме A-RTOA перед переходом к шагу 6 выполняется шаг 5.

Выбор кандидата был описан в предыдущих шагах 1-4. Количество выбранных признаков теперь определяется с использованием адаптивной стратегии на основе взаимной информации. Определение количества признаков, которые будут выбраны как 's' для каждого кандидата $k = 1, 2, \dots, n_c$.

Для каждого кандидата $k = 1, 2, \dots, n_c$ выполняется расчет количества выбранных признаков: $x_k = |v_k^{t+1}|$

Если $x_k = s$, то:

никаких изменений v_k^{t+1} .

Если $x_k < s$, то:

- а) Найти МІ для невыбранных объектов (объектов j , которые имеют $v_{k,j}^{t+1} = 0$).
- б) Добавить « $s - x_k$ » признаки, имеющие наивысшее значение МІ v_k^{t+1} (изменяя соответствующие $v_{kj} = 1$).

Если $x_k > s$, то:

- а) Найти МІ для выбранных признаков (признаков j , которые имеют $v_{k,j}^{t+1} = 1$).
- б) Удалить « $x_k - s$ » признаки, имеющие самые низкие значения МІ v_k^{t+1} (изменив соответствующие $v_k^{t+1} = 0$).

Шаг 6: Обновление глобального лучшего кандидата ($v_{\text{лучший}}$). На этом этапе можно обновить лучшего кандидата $v_{\text{лучший}}^t$.

Найти пригодность для v_k^{t+1} : пригодность (v_k^{t+1}) = Классификатор(v_k^{t+1}), $\forall k = 1, 2, 3 \dots n_c$

Определить кандидата $v_{k(\text{макс})}^{t+1}$, имеющего наибольшую точность среди $\{v_k^{t+1}: k = 1, 2, 3, \dots n_c\}$, то есть пригодность ($v_{k(\text{макс})}^{t+1}$) = $\max_{1 \leq k \leq n_c}$ пригодность (v_k^{t+1})

Правило обновления:

$$v_{\text{лучший}}^{t+1} = \begin{cases} v_{k(\text{макс})}^{t+1}, & \text{Если пригодность}(v_{k(\text{макс})}^{t+1}) > \text{пригодность}(v_{\text{лучший}}^t) \\ v_{\text{лучший}}^t, & \text{В противном случае} \end{cases}$$

Шаг 7: Повторение шагов 3–6, пока не будет выполнен критерий остановки.

Шаг 8: Возвращение выбранных характеристик $v_{\text{лучший}}$ набора данных для подготовки модели NIDS.

4. Реализация RTOA и A-RTOA

Для реализации RTOA и A-RTOA используется набор данных UNSW-NB15, который широко применяется в исследованиях, связанных с обнаружением сетевых вторжений.

4.1 Описание набора данных UNSW-NB15

Набор данных UNSW-NB-15 [40] содержит в общей сложности сорок девять признаков. Каждая запись классифицируется как невинноносная и безопасная, либо как принадлежащая конкретному типу атак. Мустафа и др. [1] предоставили полное описание всех 49 признаков, включенных в набор. Признаки подразделяются на шесть основных групп: поток, базовые, содержимое, время, дополнительные признаки и помеченные признаки.

Такие столбцы, как порт назначения, порт источника, IP-адрес назначения и IP-адрес источника, были исключены из стандартных наборов данных для обучения и тестирования. Рассматриваемый набор данных UNSW-NB15 состоял из 45 столбцов (с 43 признаками, категорией атак и меткой класса), как указано в табл. 1. Кроме того, атаки в наборе данных классифицируются по 9 различным семействам атак: Exploit, Fuzzers, Reconnaissance, Backdoor, Analysis, Shellcode, Worm, DoS (отказ в обслуживании), Generic [1].

Табл. 1. Характеристики набора данных UNSW-NB15.

Table 1. Features of UNSW-NB15 dataset.

Этикетка	Название функции	Этикетка	Название функции	Этикетка	Название функции	Этикетка	Название функции
f_1	id	f_{13}	sload	f_{25}	tcprtt	f_{37}	ct_dst_src_ltm
f_2	dur	f_{14}	dload	f_{26}	synack	f_{38}	is_ftp_login

f_3	proto	f_{15}	sloss	f_{27}	ackdat	f_{39}	ct_ftp_cmd
f_4	service	f_{16}	dloss	f_{28}	smean	f_{40}	ct_flw_http_mthd
f_5	state	f_{17}	sinpkt	f_{29}	dmean	f_{41}	ct_src_ltm
f_6	spkts	f_{18}	dinpkt	f_{30}	trans_depth	f_{42}	ct_srv_dst
f_7	dpkts	f_{19}	sjit	f_{31}	response_body_len	f_{43}	is_sm_ips_ports
f_8	sbytes	f_{20}	djit	f_{32}	ct_srv_src	f_{44}	attack_cat
f_9	dbytes	f_{21}	swin	f_{33}	ct_state_ttl	f_{45}	Label
f_{10}	rate	f_{22}	stepb	f_{34}	ct_dst_ltm	---	---
f_{11}	sttl	f_{23}	dtepb	f_{35}	ct_src_dport_ltm	---	---
f_{12}	dttl	f_{24}	dwin	f_{36}	ct_dst_sport_ltm	---	---

После сбора данных они проходят фазу предварительной обработки, где из строятся репрезентативные выборки или образцы (samples) для обеспечения пригодности для последующего анализа. В любой модели машинного обучения предварительная обработка имеет решающее значение для удаления из необработанных данных ошибок и преобразования сетевого трафика в пригодный для использования готовый для обучения модели формат. Исходный набор данных UNSW-NB15 содержит в общей сложности 175341 обучающий образец. Этот набор, который включает девять различных семейств атак, демонстрирует значительный дисбаланс классов, как между атакующими посылками и не атакующими, так и внутри самих выделенных девяти категорий атак. Распределение обучающих и тестовых образцов показывает непропорциональное представительство по классам.

Для устранения этого дисбаланса применяется гибридная техника повторной выборки [48], балансирующая распределение классов перед передачей данных на фазы обучения и тестирования. Для бинарной классификации (атака/не атака) набор данных балансируется путем выбора 131742 образцов нормального поведения и 131742 образцов атаки, что дает в общей сложности 263484 образца, из которых затем 70% выделяются для обучения, а остальные 30% для проверки его проведения. Для многоклассовой классификации используется в общей сложности 146380 образцов, с равным количеством образцов (14638), выбранных для каждого типа атаки, чтобы обеспечить единообразное представление по классам, снова разделенных на 70% для обучения и 30% для проверки.

4.2 Внедрение механизма выбора признаков в NIDS

Следующим важным шагом является выбор признаков с помощью предлагаемых алгоритмов. Архитектура для интеграции предлагаемого метода выбора признаков в NIDS представлена на рис. 1.

Полученные выбранные признаки, извлеченные с помощью методов RTOA и A-RTOA, впоследствии оцениваются с помощью нескольких классификаторов, включая CatBoost, XGBoost, KNN, дерево решений и случайный лес, с использованием различных показателей производительности, таких как правильность (accuracy), точность (precision), полнота (recall) и F1-мера.

Метрики производительности предлагаемых методов сравниваются по разным классификаторам как для бинарных, так и для многомаркированных классов. Ниже сравнивается производительность предлагаемого подхода и других методов. Система NIDS готовится к использованию с учётом оптимальных функций, определенных с помощью лучшего предлагаемого метода A-RTOA. Наконец, чтобы различать нормальное поведение и определить специфическую природу атаки, система NIDS определяет оптимальные характеристики потоков пакетов и использует модель, созданную с учетом этих характеристик.

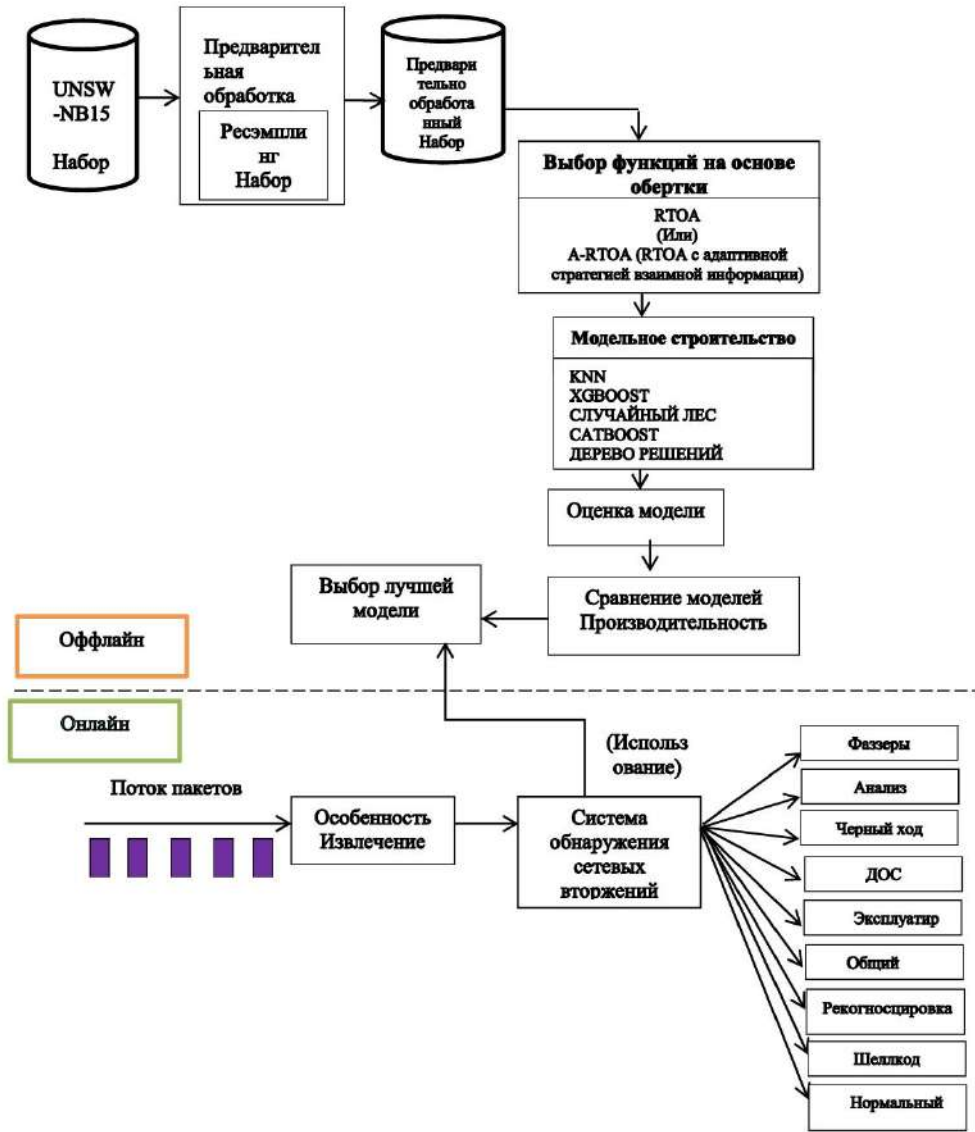


Рис. 1. Интеграция предлагаемого метода выбора функций в систему обнаружения сетевых вторжений (NIDS).

Fig. 1. Integration of the Proposed Feature Selection Method in the Network Intrusion Detection System (NIDS).

5. Результаты и обсуждение

Как обсуждалось ранее, набор данных UNSW-NB15 подвергается предварительной обработке, которая на основе предложенных алгоритмов позволяет выделить оптимальные признаки. Все эксперименты проводились на аппаратуре с 16,0 ГБ ОЗУ и процессором Intel® Core™ i7-9700F с тактовой частотой 3,00 ГГц.

Для обоих наших экспериментов в качестве классификатора используется случайный лес. Правильность модели определяется для функции пригодности, при этом значения параметров устанавливаются как $n_c = 10$, $\alpha = 0.7$, $\beta = 0.3$, а максимальное количество

итераций устанавливается равным 100. Первоначально ставится эксперимент, в котором представленный алгоритм RTOA выполняет шаги от первого до восьмого, пропуская шаг 5. Завершая шаг 8, RTOA генерирует 26 оптимальных признаков, показанных в табл. 2. Пояснения к индексам f_i ($i = 1, 2, \dots, D$) даны в табл. 1.

В эксперименте с алгоритмом A-RTOA количество выбираемых признаков, обозначенное как s , устанавливается разработчиком сети. Правильность модели с использованием случайного леса графически показана на рис. 2 для различных значений s от 1 до 43. Рис. 2 показывает эффективность A-RTOA, поскольку правильность для $s = 9$ почти такая же, как для случаев $s > 9$. Это означает, что при выборе только 9 признаков с использованием алгоритма A-RTOA вместо использования большего количества или даже всех признаков производительность сохраняется без значительной потери правильности, что позволяет быстрее проводить правильную детекцию. В конце шага 8 алгоритм A-RTOA выдает 9 оптимальных признаков, показанных в табл. 2.

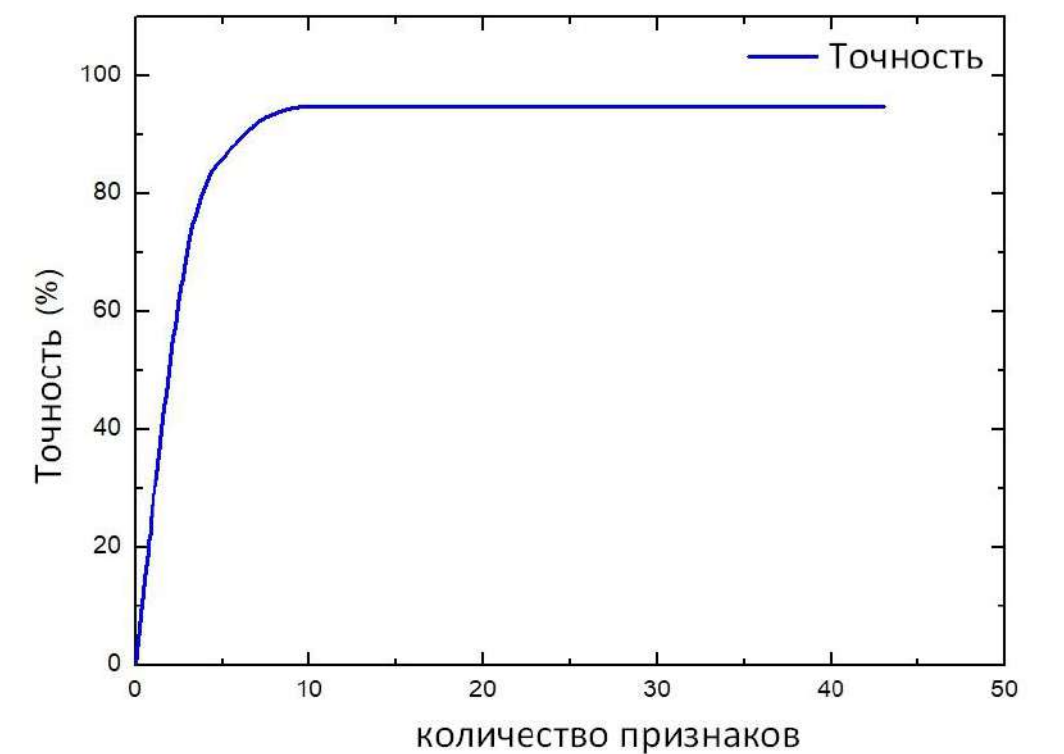


Рис. 2. Сравнение влияния количества признаков на правильность классификации.
Fig. 2. Comparison effect of feature selection on classification accuracy.

Табл. 2. Признаки, выбранные алгоритмами.
Table 2. Selected features by proposed feature selection algorithms.

Предлагаемые методы выбора признаков	Количество признаков	Признаки
RTOA	26	$f_1, f_3, f_5, f_6, f_8, f_9, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}, f_{18}, f_{19}, f_{20}, f_{21}, f_{22}, f_{23}, f_{24}, f_{25}, f_{26}, f_{28}, f_{29}, f_{30}, f_{31}$
A-RTOA	9	$f_1, f_5, f_{10}, f_{13}, f_{25}, f_{29}, f_{34}, f_{41}, f_{43}$

На рис. 3 показано сравнение времени обучения при использовании всех признаков, подмножества из 26-ти признаков и 9-ти оптимальных признаков. Результаты показывают значительное сокращение времени обучения при меньшем количестве признаков, что подчеркивает эффективность выбора признаков.

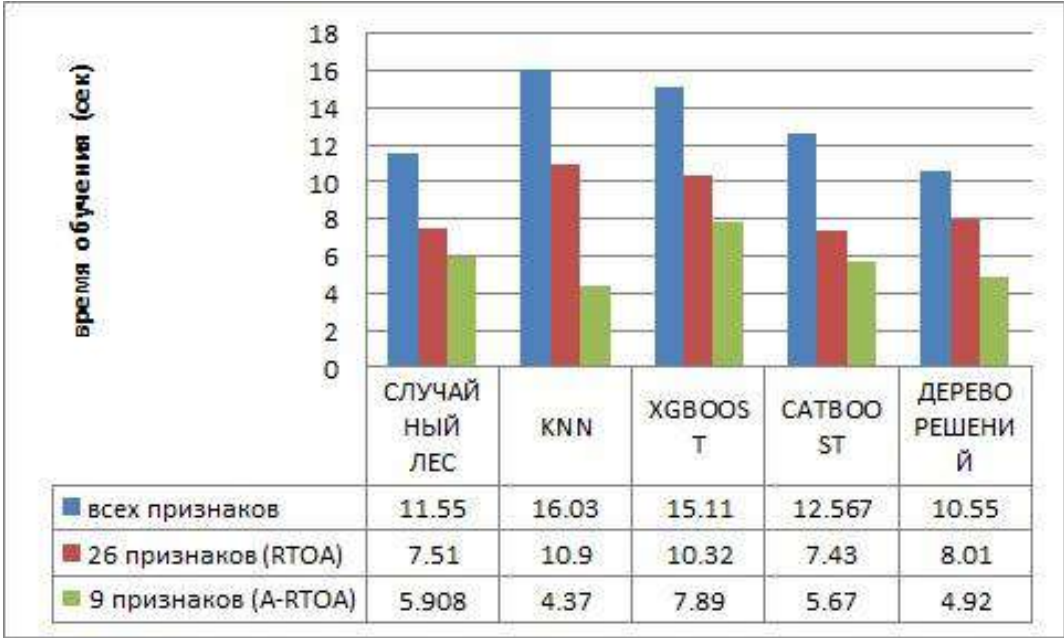


Рис. 3: Сравнение времени обучения для всех признаков, для отобранных 26 признаков и 9 оптимальных признаков.
Fig. 3. Comparisons of training time for all features, 26 and optimal 9 features.

Для сравнения производительности, чтобы вычислить оценочные метрики, мы использовали классификаторы CATBoost, XGBoost, KNN, дерево решений и случайный лес. Метрики производительности бинарной классификации модели NIDS, построенные по выбранным признакам RTOA (26 признаков), A-RTOA (9 признаков) и всем признакам, показаны в табл. 3, табл. 4 и табл. 5 соответственно. Результаты демонстрируют более высокие показатели производительности алгоритма A-RTOA, особенно при сравнении с классификатором случайного леса.

Как указано в табл. 4 и 5, девять выбранных признаков имеют производительность, эквивалентную применению всех признаков в бинарной классификации, что достигается с помощью случайного леса с точностью 94,69% и 94,73%. Аналогично, приведенные выше результаты наглядно иллюстрируют эффективность выбора признаков в сохранении точности при уменьшении размерности.

Чтобы продемонстрировать эффективность предлагаемых алгоритмов RTOA и A-RTOA, мы сравниваем их с ныне используемыми подходами. Результаты эксперимента по бинарной классификации сравниваются с существующими методами классификации и кластеризации из литературы (такими как наивный байесовский, NSENet, AdaBoost, NB-SVM и т. д.) с использованием всего набора признаков, как показано в табл. 6. Она демонстрирует эффективность предлагаемого нами метода по сравнению с современными подходами, которые используют все признаки. Правильность, достигнутая с использованием 9 признаков, выше, чем с 26 признаками, и оба выше точности, полученной с использованием полного набора признаков. Это ясно подчеркивает эффективность нашей стратегии выбора

признаков. Кроме того, производительность алгоритмов RTOA и A-RTOA сравнивается с существующими подходами к оберткам и фильтрам (такими как алгоритмы PSO, GWO, FFA, GA и другими), предложенными разными авторами, как показано в табл. 7.

Табл. 3. Оценка производительности бинарной классификации с использованием оптимальных признаков (числом 26), выбранных алгоритмом RTOA.

Table 3. Performance evaluation for binary classification using optimal features (26) selected by RTOA.

Метрики	CATBOOST	XGBOOST	КНН	ДЕРЕВО РЕШЕНИЙ	СЛУЧАЙНЫЙ ЛЕС
Правильность (%)	87.26	88.22	87.31	93,55	94.12
Точность (%)	87,71	88.93	86.75	93.28	93,87
Отзывать (%)	87.26	88.22	87.31	93,55	94.12
Оценка F1 (%)	87.17	88.13	86.48	93.35	93.92

Табл. 4: Оценка производительности бинарной классификации с использованием оптимальных признаков (числом 9), выбранных адаптивным алгоритмом A-RTOA.

Table 4. Performance evaluation for binary classification using optimal features (9) selected by A-RTOA.

Метрики	CATBOOST	XGBOOST	КНН	ДЕРЕВО РЕШЕНИЙ	СЛУЧАЙНЫЙ ЛЕС
Правильность (%)	88.22	89.12	90,75	94.18	94.69
Точность (%)	88.73	89.97	90,69	93.97	94.51
Отзывать (%)	88.22	89.12	90,75	94.18	94.69
Оценка F1 (%)	88.18	89.05	90.26	94.01	94,54

Табл. 5: Оценка эффективности бинарной классификации с использованием всех признаков.

Table 5. Performance evaluation for binary classification using all features.

Метрики	CATBOOST	XGBOOST	КНН	ДЕРЕВО РЕШЕНИЙ	СЛУЧАЙНЫЙ ЛЕС
Правильность (%)	88.68	89.69	90.32	94.24	94.73
Точность (%)	89.16	90.44	90.21	94.02	94,54
Отзывать (%)	88.68	89.69	90.32	94.24	94.73
Оценка F1 (%)	88.69	89,68	89,75	94.07	94.58

Табл. 7 показывает эффективность предлагаемых нами методов RTOA и A-RTOA. Видно, что адаптивный алгоритм A-RTOA превосходит как алгоритм RTOA, так и другие современные методы с точки зрения как правильности, так и по количеству признаков.

Поскольку UNSW-NB15 представляет собой набор данных с многоклассовой разметкой, для многоклассовой классификации используется предварительно обработанный и нормализованный набор данных. Метрика производительности (правильность) предлагаемых нами RTOA (26 признаков) и A-RTOA (9 признаков) для многоклассовой классификации с использованием классификатора случайного леса сравнивается с методами из литературы [9, 26, 37 и 39], как показано в табл. 8. Видно, как предлагаемые методы повышают правильность и эффективность NIDS в сравнении с существующими подходами.

На основании результатов и обсуждений можно сделать вывод, что предлагаемый метод RTOA работает лучше с првильностью 94,12% по сравнению с существующими алгоритмами (68,18% против 93,01% с более чем 9 признаками) из литературы [11, 15, 17, 22, 24, 27, 29, 34-38]. Более того, адаптивная стратегия A-RTOA достигает наилучшей производительности с точностью 94,69%, превосходя RTOA (94,12%) и другие перечисленные современные методы для бинарной классификации. Этот вывод также распространяется на многоклассовую классификацию. Адаптивная стратегия A-RTOA достигает наилучшей производительности со средней правильностью 94,03%, превосходя RTOA со средней правильностью 93,92% для многоклассовой классификации.

6. Заключение

В этой статье предложена новый метаэвристическая оптимизация алгоритма «черепаха и заяц» (RTOA), а также адаптивная стратегия на основе взаимной информации с этим алгоритмом оптимизации, называемая A-RTOA. Эти методы выполняют отбор признаков при обнаружении сетевых вторжений. Методы реализованы с использованием набора данных UNSW-NB15. NIDS построены на двух оптимальных наборах признаков: 26 признаков, выбранных с помощью RTOA, и 9 признаков, выбранных с помощью A-RTOA. Производительность NIDS определяется и сравнивается с производительностью других алгоритмов NIDS. Сравнение бинарной классификации, многоклассовой классификации и важность сокращенных оптимальных признаков проведено в разделе результатов и показывает, что RTOA превосходит существующие методы отбора признаков, в то время как A-RTOA работает еще лучше, чем RTOA. Система NIDS, созданная с использованием A-RTOA, надежна и имеет большое значение для достижения надежного и эффективного обнаружения сетевых вторжений. Предложенные методы усиливают сетевую защиту и помогают справиться с проблемами, связанными с растущими киберугрозами. В будущем можно исследовать дальнейшие усовершенствования и методы для развития полученных результатов и улучшения возможностей многоклассовой классификации обнаружения сетевых вторжений.

Табл. 6. Сравнение бинарных классификаторов с несколькими методами классификации/кластеризации, а также RTOA и A-RTOA.

Table 6. Comparison of binary classifiers with a few Classification/Clustering methods and RTOA/A-RTOA.

	Методы классификации/кластеризации	Количество функций	Точность (%)
В литературе	Слияние NSENet и LSTM на основе SENet [12]	Все функции	82.14
	Дерево решений [14]		90.15
	АдаБуст [14]		90,51
	Дерево усиления градиента [14]		87,56
	Многослойный перцептрон [14]		84.11
	AdaBoost, долго-кратковременная память [14]		87.90
	Закрытый рекуррентный блок [14]		82.87
	Модели ансамбля [17]		90.98
	Нечеткие С-средние [18]		73,62
	Дерево решений [45]		85,56
	Логистическая регрессия [45]		83.15
	Наивный Байес [45]		82.07
	Искусственная нейронная сеть [45]		81.34
	Кластеризация ожиданий-максимизации [45]		78.47
	Модель ансамбля Голосование [47]		89.29
	Дерево регрессии с градиентным усилением [48]		91.31
Предлагаемый метод RTOA	NB-SVM [49]	26	93,75
	CATBOOST		87.26
	XGBOOST		88.22
	KNN		87.31
	ДЕРЕВО РЕШЕНИЙ		93,55
Предлагаемый метод A-RTOA	СЛУЧАЙНЫЙ ЛЕС	9	94.12
	CATBOOST		88.22
	XGBOOST		89.12
	KNN		90,75
	ДЕРЕВО РЕШЕНИЙ		94.18
	СЛУЧАЙНЫЙ ЛЕС		94.69

Табл. 7. Сравнение различных существующих методов и предлагаемых методов RTOA и A-RTOA.
Table 7. Comparison between different existing methods and our proposed methods RTOA and A-RTOA.

№	Литература	Метод выбора характеристик	Число признаков	Используемые классификаторы	Точность (%)	
1	Диша и др. [14]	Подход фильтра – взвешенный случайный лес на основе примесей Джини (ГИВРФ)	20	Дерево решений	93.01	
				АдаБуст	90,51	
				GBT	87.08	
				MLP	87.26	
				LSTM	88.99	
				GRU	90.11	
2	Алаззам и др. [19]	Подход обертки – оптимизатор, вдохновленный Pigeon (PIO): Sigmoid_PIO	14	Дерево решений (DT)	91.30	
3	Мегантара и Ахмад [21]	Подход обертки: гибридный метод машинного обучения	11	Дерево решений	91.86	
4	Девпрасад и др. [28]	Подход «Фильтр и обертка»: алгоритмы χ^2 и «летучая мышь»	9	Классификаторы DT и опорных векторных машин	89.43	
5	Касонго и др. [30]	Нормализация функций с помощью XGBoost	19	Дерево решений	90,85	
6	Альмомани и др. [35]	Подход обертки: PSO, GWO, FFA и GA, биоинспирированные алгоритмы и MI	13	Дерево решений	89,58	
			30		90.48	
7	Белух и др. [37]	Выбор признаков на основе фильтра (IG) с двухэтапным подходом к классификации	20	Уменьшение ошибок обрезки дерева (RepTree)	88.95	
8	Мустафа и др. [50]	Фильтрационный подход (PCA) с геометрическим анализом площади (GAA)	15	Дерево решений	92.80	
9	Мустафа и др. [51]	Гибридный выбор признаков с использованием ассоциативных правил	11	Кластеризация ожиданий-максимизации	77.2	
				Логистическая регрессия	83.0	
				Наивный Байес	79,5	
10	Хаммасси и др. [52]	Подход обертка: Генетический алгоритм и логистическая регрессия	20	Классификатор C4.5	81.42	
11	Сети и др. [53]	Подход, основанный на обучении с подкреплением	19	Случайный лес (RF)	82.12	
				АдаБуст (ADB)	85,61	
				Гауссовский наивный байесовский алгоритм (GNB),	70,62	
				k-ближайших соседей (KNN)	78.47	
				Квадратичный дискриминантный анализ (QDA)	68.18	
				RF+ADB+QDA	85.09	
12	Кумар и др. [54]	Метод, основанный на правилах	13	Дерево решений	84.83	
				Дерево решений C5	90,74	
13	Предлагаемый подход к обертке: RTOA		26	CATBOOST	87.26	
				XGBOOST	88.22	
				KNN	87.31	
				ДЕРЕВО РЕШЕНИЙ	93,55	
				СЛУЧАЙНЫЙ ЛЕС	94.12	
14	Предлагаемый подход к обертке: RTOA с адаптивной стратегией (A-RTOA)		9	CATBOOST	88.22	
				XGBOOST	89.12	
				KNN	90,75	
				ДЕРЕВО РЕШЕНИЙ	94.18	
				СЛУЧАЙНЫЙ ЛЕС	94.69	

Табл. 8: Сравнение показателей производительности (точность, %) для многоклассовой классификации: предлагаемые методы и данные из литературы.
Table 8. Comparison of Performance Metrics (Accuracy %) for Multi-Label Classification: Proposed Methods vs. Literature.

№	Характер атак	Халладжи и др., [9] Многокомпонентная нейронная сеть на основе ансамбля	Мефтах и др. [26] C5.0 Дерево решений	Сети и др., [37] Случайный лес	Мустафа и др., [39] Наивный Байес	Предлагаемые алгоритмы	
						RTOA (26 функций) Случайный лес	A-RTOA (9 функций) Случайный лес
1	Анализ	97.47	21.35	84.67	0	97.76	97.92
2	Черный ход	35.56	17.75	83.53	20	97.63	97.62
3	ДОС	94.40	10.38	92.12	71.1	91.74	92.33
4	Эксплуатация	70.17	96.68	79.21	54.6	86.48	89.34
5	Фаззеры	86.08	79.10	93.43	33.2	87.66	90.35
6	Общий	77.27	98.72	96.37	0	93.84	95.21
7	Нормальный	32.35	93.73	-	94.3	96.99	97.67
8	Рекогносцировка	36.88	76.29	89.45	69.9	95.78	96.46
9	Шеллкод	81.75	84.02	92.79	0	96.80	91.67
10	Черви	86.96	63.63	65.31	0	94.53	91.68

Список литературы / References

[1]. Moustafa, Nour, and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In 2015 military communications and information systems conference (MilCIS), pp. 1-6. IEEE, 2015.

[2]. Choudhary, Sarika, and Nishtha Kesswani. Analysis of KDD-Cup'99, NSL-KDD and UNSW-NB15 datasets using deep learning in IoT. Procedia Computer Science 167 (2020): 1561-1573.

[3]. Selvakumar, B., and Karuppiiah Muneeswaran. Firefly algorithm based feature selection for network intrusion detection. Computers & Security 81 (2019): 148-155.

[4]. Bhuvaneswari, T., M. Chengathir Selvi, R. Naga Priyadarsini, U. Eswaran, and RK Ramesh Babu. Feature selection with mutual information based cuckoo search optimization for Parkinson's disease prediction. NeuroQuantology 20, no. 10 (2022): 1296.

[5]. Rana, Pratip, Phuc Thai, Thang Dinh, and Preetam Ghosh. Relevant and non-redundant feature selection for cancer classification and subtype detection. Cancers 13, no. 17 (2021): 4297.

[6]. Tahir, Mahjabeen, et al. A novel approach for handling missing data to enhance network intrusion detection system. Cyber Security and Applications 3 (2025): 100063.

[7]. Saheed, Yakub Kayode, and Sanjay Misra. A voting gray wolf optimizer-based ensemble learning models for intrusion detection in the Internet of Things. International Journal of Information Security 23.3 (2024): 1557-1581.

[8]. Zhu, Jingyi, and Xiufeng Liu. An integrated intrusion detection framework based on subspace clustering and ensemble learning. Computers and Electrical Engineering 115 (2024): 109113.

[9]. Hallaji, Ehsan, Roozbeh Razavi-Far, and Mehrdad Saif. Expanding analytical capabilities in intrusion detection through ensemble-based multi-label classification. Computers & Security 139 (2024): 103730.

[10]. Li, Shaoqin, Zhendong Wang, Shuxin Yang, Xiao Luo, Daojing He, and Sammy Chan. Internet of Things intrusion detection: Research and practice of NSENet and LSTM fusion models. Egyptian Informatics Journal 26 (2024): 100476.

[11]. Disha, Raisa Abedin, and Sajjad Waheed. Performance analysis of machine learning models for intrusion detection system using Gini Impurity-based Weighted Random Forest (GIWRF) feature selection technique. Cybersecurity 5.1 (2022): 1.

[12]. Keserwani, Pankaj Kumar, Mahesh Chandra Govil, and Emmanuel S. Pilli. An effective NIDS framework based on a comprehensive survey of feature optimization and classification techniques. Neural Computing and Applications 35.7 (2023): 4993-5013.

- [13]. Yousefnezhad, Maryam, Javad Hamidzadeh, and Mohammad Aliannejadi. Ensemble classification for intrusion detection via feature extraction based on deep Learning. *Soft Computing* 25.20 (2021): 12667- 12683.
- [14]. Kabilan, N., Vinayakumar Ravi, and V. Sowmya. Unsupervised intrusion detection system for in-vehicle communication networks. *Journal of Safety Science and Resilience* 5.2 (2024): 119-129.
- [15]. Alazzam, Hadeel, Ahmad Sharieh, and Khair Eddin Sabri. A feature selection algorithm for intrusion detection system based on pigeon inspired optimizer. *Expert systems with applications* 148 (2020): 113249.
- [16]. Feroz Khan, A. B., and Anandharaj, G. A Multi-layer Security approach for DDoS detection in Internet of Things. *International Journal of Intelligent Unmanned Systems* 9, no. 3 (2021): 178-191.
- [17]. Megantara, Achmad Akbar, and Tohari Ahmad. A hybrid machine learning method for increasing the performance of network intrusion detection systems. *Journal of Big Data* 8, no. 1 (2021): 142.
- [18]. Almiani, Muder, Alia AbuGhazleh, Amer Al-Rahayfeh, Saleh Atiewi, and Abdul Razaque. Deep recurrent neural network for IoT intrusion detection system. *Simulation Modelling Practice and Theory* 101 (2020): 102031.
- [19]. Bhattacharya, Sweta, Praveen Kumar Reddy Maddikunta, Rajesh Kaluri, Saurabh Singh, Thippa Reddy Gadekallu, Mamoun Alazab, and Usman Tariq. A novel PCA-firefly based XGBoost classification model for intrusion detection in networks using GPU. *Electronics* 9, no. 2 (2020): 219.
- [20]. Moualla, Soulaïman, Khaldoun Khorzom, and Assef Jafar. Improving the Performance of Machine Learning-Based Network Intrusion Detection Systems on the UNSW-NB15 Dataset. *Computational Intelligence and Neuroscience* 2021, no. 1 (2021): 5557577.
- [21]. Saheed, Yakub Kayode., and Micheal Olaolu Arowolo. Efficient cyber attack detection on the internet of medical things-smart environment based on deep recurrent neural network and machine learning algorithms. *IEEE Access* 9 (2021): 161546-161554.
- [22]. Devprasad, Kayathri Devi, Sukumar Ramanujam, and Suresh Babu Rajendran. Context adaptive ensemble classification mechanism with multi-criteria decision making for network intrusion detection. *Concurrency and Computation: Practice and Experience* 34, no. 21 (2022): e7110.
- [23]. Shiravani, Anita, Mohammad Hadi Sadreddini, and Hassan Nosrati Nahook. Network intrusion detection using data dimensions reduction techniques. *Journal of Big Data* 10, no. 1 (2023): 27.
- [24]. Kasongo, Sydney M., and Yanxia Sun. Performance analysis of intrusion detection systems using a feature selection method on the UNSW-NB15 dataset. *Journal of Big Data* 7, no. 1 (2020): 105.
- [25]. Kumar, Vikash, Ditipriya Sinha, Ayan Kumar Das, Subhash Chandra Pandey, and Radha Tamal Goswami. An integrated rule based intrusion detection system: analysis on UNSW-NB15 data set and the real time online dataset. *Cluster Computing* 23 (2020): 1397-1418.
- [26]. Meftah, Souhail, Tajjeeddine Rachidi, and Nasser Assem. Network based intrusion detection using the UNSW-NB15 dataset. *International Journal of Computing and Digital Systems* 8, no. 5 (2019): 478-487.
- [27]. Almomani, Omar. A feature selection model for network intrusion detection system based on PSO, GWO, FFA and GA algorithms. *Symmetry* 12, no. 6 (2020): 1046.
- [28]. Ahmad, Zeeshan, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, and Farhan Ahmad. Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies* 32, no. 1 (2021): e4150.
- [29]. Belouch, Mustapha, Salah El Hadaj, and Mohamed Idhammad. A two-stage classifier approach using reptree algorithm for network intrusion detection. *International Journal of Advanced Computer Science and Applications* 8, no. 6 (2017).
- [30]. Moustafa, Nour, and Jill Slay. The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set. *Information Security Journal: A Global Perspective* 25, no. 1-3 (2016): 18-31.
- [31]. Swami, Rochak, Mayank Dave, and Virender Ranga. Voting-based intrusion detection framework for securing software-defined networks. *Concurrency and computation: practice and experience* 32.24 (2020): e5927.
- [32]. Tama, Bayu Adhi, and Kyung-Hyune Rhee. An in-depth experimental study of anomaly detection using gradient boosted machine. *Neural Computing and Applications* 31 (2019): 955-965.
- [33]. Gu, Jie, and Shan Lu. An effective intrusion detection approach using SVM with naïve Bayes feature embedding. *Computers & Security* 103 (2021): 102158.
- [34]. Moustafa, Nour, Jill Slay, and Gideon Creech. Novel geometric area analysis technique for anomaly detection using trapezoidal area estimation on large-scale networks. *IEEE Transactions on Big Data* 5.4 (2017): 481-494.

- [35]. Moustafa, Nour, and Jill Slay. A hybrid feature selection for network intrusion detection systems: Central points. arXiv preprint arXiv:1707.05505 (2017).
- [36]. Khammassi, Chaouki, and Saoussen Krichen. A GA-LR wrapper approach for feature selection in network intrusion detection. *computers & security* 70 (2017): 255-277.
- [37]. Sethi, Kamalakanta, E. Sai Rupesh, Rahul Kumar, Padmalochan Bera, and Y. Venu Madhav. A context-aware robust intrusion detection system: a reinforcement learning-based approach. *International Journal of Information Security* 19 (2020): 657-678.
- [38]. Kumar, V., D. Sinha, A. K. Das, S. C. Pandey, and R. T. Goswami. An integrated rule based Intrusion Detection System: Analysis on UNSW-NB15 Data Set and the real time online dataset-cluster computing. SpringerLink (2019).
- [39]. Moustafa, Nour, and Jill Slay. The significant features of the UNSW-NB15 and the KDD99 data sets for network intrusion detection systems. In 2015 4th international workshop on building analysis datasets and gathering experience returns for security (BADGERS), pp. 25-31. IEEE, 2015.
- [40]. <https://research.unsw.edu.au/projects/unswnb15-dataset>.
- [41]. Breiman, Leo. Random forests. *Machine learning* 45 (2001): 5-32.
- [42]. Dorogush, Anna Veronika, Vasily Ershov, and Andrey Gulin. CatBoost: gradient boosting with categorical features support. arXiv preprint arXiv:1810.11363 (2018).
- [43]. Chen, Tianqi, and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785-794. 2016.
- [44]. Wang, Hui. Nearest neighbours without k: a classification formalism based on probability. Faculty of Informatics, University of Ulster (2002).
- [45]. Jung, Alexander. *Machine learning: the basics*. Springer Nature, 2022.
- [46]. Peter Gammie, The Tortoise and Hare Algorithm, 2015, [online] Available: <http://isa-afp.org/entries/TortoiseHare.html>.
- [47]. MI Hoque, Nazrul, Dhruba K. Bhattacharyya, and Jugal K. Kalita. MIFS-ND: A mutual information-based feature selection method. *Expert systems with applications* 41, no. 14 (2014): 6371-6385.
- [48]. Srinivasan, Manohar, and Narayanan Chidambaram Senthil kumar. Class imbalance data handling with optimal deep learning-based intrusion detection in IoT environment. *Soft Computing* 28, no. 5 (2024): 4519-4529.

Информация об авторах / Information about authors

Тамиларасан БХУВАНЕШВАРИ – доцент кафедры компьютерных наук и инженерии инженерного колледжа Мепко Schlenk с 2021 года. Ее научные интересы – машинное обучение, обнаружение вторжений, методы оптимизации.

Thamilarasan BHUVANESWARI – Assistant professor, Department of computer science and engineering of Mepco Schlenk engineering college since 2021. Her research interests are machine learning, intrusion detection, optimization techniques.

Саундар Катхавараян РУБА – Dr. Sci., доцент (старший класс) кафедры компьютерных наук и инженерии инженерного колледжа Мепко Шленк с 2021 года. Область научных интересов: классификация текстур, субтитры к видео, агрегация данных, обнаружение теней, DDoS-атаки и обнаружение вторжений.

Soundar Kathavarayan RUBA – Dr. Sci., Associate professor (Sr. Grade), Department of computer science and engineering of Mepco Schlenk engineering college since 2021. Research interests: texture classification, video captioning, data aggregation, shadow detection, DDoS attack and intrusion detection.

Гуру Секар Рамакришнан ЧАНДРА – доцент (старший класс) кафедры математики инженерного колледжа Мепко Schlenk с 2017 года. Область научных интересов: численные методы, методы оптимизации и обработки изображений.

Guru Sekar Ramakrishnan CHANDRA – Assistant professor (Sr. Grade), Department of mathematics of Mepco Schlenk engineering college since 2017. Research interests: numerical methods, optimization techniques and image processing.



Определение неточностей в работе некоторых специализированных цикловых оптимизаций в компиляторе LCC для архитектуры «Эльбрус»

¹ А.В. Ермолицкий, ORCID: 0009-0008-2407-7420 <era@mcst.ru>

^{1,2} Д.Н. Левченко, ORCID: 0009-0004-6744-9889 <levchenko_d@mcst.ru>

^{1,2} М.И. Нейман-заде, ORCID: 0000-0002-4250-9724 <muradnz@mcst.ru>

¹ АО «МЦСТ», 117437, Москва, ул. Профсоюзная, д. 108.

² Московский физико-технический институт (национальный исследовательский университет), Россия 117303, Москва, ул. Керченская, д.1 А, корп. 1.

Аннотация. Работа посвящена развитию смешанных методов анализа неточностей, возникающих при проведении компиляторных оптимизаций. Развитие этих методов важно для процессоров с широким командным словом (VLIW), построенных на архитектуре «Эльбрус» со статическим планированием. Проанализированы существующие подходы к выявлению неточностей в работе оптимизаций, выделены их недостатки. Авторами разработан метод обнаружения неточностей в работе двух важных для VLIW оптимизаций: конвейеризации циклов с аппаратной поддержкой (overlap) и оптимизации выноса участков цикла с малой вероятностью исполнения в создаваемый охватывающий цикл (nesting). Метод реализуется посредством инструментирования циклов в пользовательской программе и получения статической информации о работе циклов от компилятора. Предложенный метод был проверен на задачах из пакетов SPEC CPU 2006 и 2017 rate в режиме base (без использования профильной информации) на ЭВМ с процессором «Эльбрус-8С», где доказал свою эффективность. Метод позволил достичь ускорения до 70.7% на отдельных задачах при расстановке подсказок к оптимизации overlap и 4.71% на задаче 520.omnetpp при расстановке подсказок к оптимизации nesting.

Ключевые слова: компиляторные оптимизации; программная конвейеризация; инструментирование кода; широкое командное слово VLIW.

Для цитирования: Ермолицкий А.В., Левченко Д.Н., Нейман-заде М.И. Определение неточностей в работе некоторых специализированных цикловых оптимизаций в компиляторе LCC для архитектуры «Эльбрус». Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 51–64. DOI: 10.15514/ISPRAS-2025-37(4)-3.

Detection of Inaccuracies in Some Specialized Loop Optimizations in the LCC Compiler for “Elbrus” Architecture

¹ A.V. Ermolitsky, ORCID: 0009-0008-2407-7420 <era@mcst.ru>

^{1,2} D.N. Levchenko, ORCID: 0009-0004-6744-9889 <levchenko_d@mcst.ru>

^{1,2} M.I. Neiman-zade, ORCID: 0000-0002-4250-9724 <muradnz@mcst.ru>

¹ AO “MCST”, 108, Profsoyuznaya str., Moscow, 117437, Russia.

² Moscow Institute of Physics and Technology (National Research University),
building 1, 1 A, Kerchenskaya st., Moscow, 117303, Russia.

Abstract. This work is devoted to the development of mixed methods for analyzing inaccuracies in compiler optimizations. The development of these methods is important for “Elbrus” microprocessors with a very long instruction word (VLIW) because of static scheduling. The article analyzes existing approaches to identifying inaccuracies in optimizations and highlights their disadvantages. In this article a method is developed for detecting inaccuracies in the work of two important for VLIW optimizations: software pipeline with hardware support (overlap) and optimization of moving unlikely executed code into new outer loop (nesting). The method is implemented by instrumenting loops in the user program and obtaining static information about loops from the compiler. The method was checked on SPEC CPU 2006 and 2017 rate suites on a computer with an Elbrus-8S processor and has proven its effectiveness. The method allows to achieve a speedup of 70.7% on test 523.xalancbmk with placing hints for overlap optimization and 4.71% on test 520.omnetpp with placing hints for nesting optimization. Tests are done in base mode without profile information.

Keywords: compiler optimizations; software pipeline; code instrumentation; VLIW.

For citation: Ermolitsky A.V., Levchenko D.N., Neiman-zade M.I. Detection of inaccuracies in specialized loop optimizations in the LCC compiler for “Elbrus” architecture. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025. pp. 51-64 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-3.

1. Введение

Особенность микропроцессоров на основе архитектур со статическим планированием заключается в том, что процесс обнаружения и использования параллелизма инструкций производится только при построении машинного кода оптимизирующим компилятором. Такой подход позволяет существенно упростить микроархитектуру процессора по сравнению с суперскалярными микропроцессорами с внеочередным исполнением, хотя и за счёт определённого усложнения оптимизирующих компиляторов.

Однако такой архитектурный подход обладает своими недостатками. В частности, компилятор (однопроходный) может производить оптимизации только на основе статической информации, используя эвристики, тогда как динамическая информация ему недоступна. Это приводит к тому, что применение компилятором некоторых оптимизирующих преобразований к коду вносит накладные расходы, которые во время исполнения могут оказаться существенными и вместо ускорения пользовательской программы, в таких случаях может происходить её замедление.

Устранить неточности в работе оптимизаций можно с помощью специальных подсказок в исходном коде. Инструмент, рассматриваемый в данной работе, позволяет определить, где и какие именно подсказки следует поставить программисту в исходном коде, чтобы избежать неточностей в работе оптимизаций на горячих участках кода.

Важная особенность рассматриваемого инструмента состоит в том, что необходимо избежать проблем, аналогичных режиму двухпроходной компиляции с использованием профильной информации. В частности, необходимости перекомпиляции и исполнения программы на тестовых данных каждый раз, когда происходит изменение кодовой базы. Поэтому в данном инструменте отчёт о найденных неоптимальностях в работе оптимизаций не подходит для использования в качестве аналога профильной информации.

2. Обзор существующих подходов

Существующие методы в поиске неточностей в работе оптимизаций компилятора делятся на три подхода:

- 1) Первый подход предполагает использование только динамической информации, полученной при исполнении программы. В данном подходе используются результаты, полученные от инструментов статистического профилирования, таких как `perf` или `drprof`. Однако инструменты профилирования программы лишь помогают определить часто исполняемые фрагменты кода, но не говорят о проблемах в работе оптимизаций, так как используют только динамическую информацию и не учитывают статическую информацию от компилятора. При работе с такими профилировщиками нужна высокая квалификация программиста, который должен знать устройство работы оптимизатора в компиляторе и самостоятельно определять проблемы в его работе. Однако программисту придётся столкнуться с рядом проблем. Во-первых, добиться нужного результата невозможно в случае, если используется компилятор с закрытым исходным кодом. Во-вторых, количество оптимизаций со сложными взаимными зависимостями в современных компиляторах исчисляется сотнями. Оптимизации могут создавать контекст для работы друг друга. Разобраться человеку без средств автоматизации оказывается сложно. В-третьих, стоит учитывать проблему доверия профилировщикам. В частности, по своему устройству, статистические профилировщики могут предоставлять лишь статическую информацию о частоте исполнения кода, не являющуюся в полной мере достоверной [1-2]. В свою очередь использование инструментальных средств профилирования осложнено тем, что они вносят существенные накладные расходы, увеличивая как размеры исполняемых файлов за счёт создания структур в статически выделенной памяти, так и времена исполнения пользовательских программ за счёт дополнительно внедрённого кода.
- 2) Второй подход опирается только на статическую информацию об исходном коде и информацию от компилятора. Он предполагает использование инструментов обзора применённых оптимизаций. Основным представителем является режим работы `opt-report`, который поддерживают все современные компиляторы ("`llvm-opt-report`" у LLVM, "`-fopt-report`" у LCC, "`-qopt-report`" у ICC, "`-fopt-info`" у GCC). Данный режим компиляции показывает программисту какие оптимизации применились и не применились, оперируя только статической информацией без исполнения скомпилированной пользовательской программы. Для анализа проблем в работе оптимизаций, полезным инструментом могут быть программы для обзора бинарных файлов, такие как CCNav [3]. Программа CCNav позволяет сопоставить бинарный файл с исходным кодом вместе с совместным отображением графа потока управления и графа вызовов. Однако данный инструмент, как и в работе с профилировщиками, полагается на знания программиста и не предоставляет никакой информации от компилятора. С учётом того, что «Эльбрус» – это архитектура с поддержкой предикатов, после работы оптимизации слияния альтернатив ветвления кода понять по ассемблерному тексту, какие оптимизации применились к конкретному циклу, очень тяжело. Именно поэтому режим "`-fort-report`" является предпочтительным. Общим недостатком этих двух инструментов является отсутствие поддержки динамической информации, поэтому данные инструменты не могут подсказать, перевешивают ли накладные расходы положительный эффект от применения оптимизации или нет.
- 3) Третий подход является комбинированным подходом, в котором инструменты полагаются и на статическую, и на динамическую информацию. Данный подход предполагает использование комплексных систем анализа производительности, таких как Intel Advisor [4]. Он использует отладочную информацию, которая предоставляется компилятором в режиме "`-O2 -g`", информацию от статистического профилирования

и информацию о событиях в процессоре. Благодаря этой информации Intel Adviser может определять проблемы с векторизацией циклов и сообщать пользователю наиболее вероятную причину отсутствия векторизации и способ исправления ситуации. Однако, данный подход лишь косвенно устанавливает причины отсутствия векторизации, но не других оптимизаций, так как это не является актуальным для процессоров с динамическим планированием.

3. Метод поиска неоптимальностей

Предлагаемый в статье метод поиска неоптимальностей относится к третьему подходу. Однако различие с уже известными подходами заключается в том, что в качестве статической информации берётся не отладочная информация о всём исходном коде, а только причины проблем применения оптимизаций к конкретному циклу. Об этих причинах может сообщить компилятор в процессе оценки возможности применения оптимизации. В данной статье рассмотрены способы определения неточностей в оптимизации конвейеризации циклов с аппаратной поддержкой (overlap) [5] и оптимизации выноса участков кода с малой вероятностью исполнения в создаваемый охватывающий цикл (nesting) [6]. Данные оптимизации могут существенно повышать производительность кода циклов при компиляции под архитектуру «Эльбрус».

3.1 Общая схема работы

Анализ неоптимальностей в работе оптимизаций проводится в два этапа.

Первый этап – это этап компиляции. На этом этапе, по указанию, сделанному пользователем, строится дополнительная структура данных, содержащая массивы дескрипторов. Также, в пользовательский код внедряются инструкции увеличения значений определённых счётчиков в указанные дескрипторы. В данной работе под дескриптором будем понимать структуру, которая описывает один цикл. В ней содержится такая информация от компилятора, как:

- привязка к файлу исходного кода;
- предсказанное эвристиками среднее число итераций;
- счётчик заходов в предцикл;
- счётчик количества итераций тела цикла;
- информация о применённых оптимизациях;
- различные служебные флаги для конкретной реализации.

Важно, что вставка дополнительного кода должна вносить как можно меньше побочных эффектов, так как целью является инструментирование работы оптимизаций. Вставка инструментирующего кода может внести возмущения, которые приведут к иному поведению работы оптимизаций. Поэтому инструментирование оптимизаций должно вносить как можно меньше вызовов библиотечных функций и полагаться на вставку линейных участков кода.

При инструментировании циклов, важно помнить, что операции Load и Store могут также вносить возмущения и приводить к иной работе оптимизаций, поэтому, необходимо стараться выносить эти операции за тело инструментируемого цикла. Этого можно достигнуть, если положить объект-счётчик, из соответствующего цикла дескриптора, на регистр перед исполнением тела цикла, а запись в объект сделать после исполнения тела цикла. Соответственно, в теле цикла должна остаться только операция ADD, которая будет вносить минимальное возмущение.

Второй этап – это этап исполнения. Пользователь должен запустить инструментированную раннее программу (без каких-либо дополнительных опций). Стоит отметить, что запуск должен проводиться на репрезентативных данных, отражающих типичные варианты использо-

вания программы. Во время этого исполнения будет набрана статистика за счёт работы операций увеличения значений счётчиков. После окончания работы пользовательской программы вызывается функция библиотеки поддержки, в которой происходит чтение дескрипторов, проведение анализа и вывод отчёта программисту.

3.2 Инструментирование оптимизации overlap

Оптимизация overlap в компиляторе LCC представляет собой программно-аппаратную конвейеризацию циклов, которая позволяет исполнять несколько итераций совместно, как показано на рис. 1б, эффективно используя аппаратные возможности архитектуры «Эльбрус». Благодаря ей, можно эффективно использовать исполняющие устройства в процессоре и увеличить степень заполнения широкой команды. Это одна из самых важных цикловых оптимизаций для архитектуры «Эльбрус», которая позволяет ускорить циклы в несколько раз. Необходимость её инструментирования вызвана тем, что, при малом числе итераций в цикле, накладные расходы могут превышать пользу, которая даёт данная оптимизация. Самые часто замеченные примеры – это неправильное предсказание количества итераций при обходе динамических структур, таких как деревья или списки.

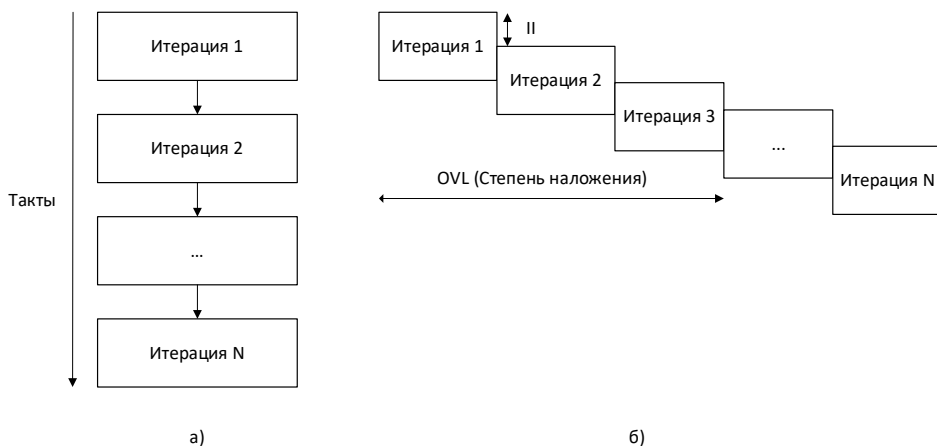


Рис. 1. а) Последовательное исполнение цикла.
б) Исполнение цикла с наложением итераций после оптимизации overlap.
Fig. 1. а) Sequential execution of a loop.
б) Loop execution with overlapping iterations after overlap optimization.

При проведении оптимизации overlap происходит создание дубликата цикла. Для одной из копий цикла производится применение оптимизации. Затем, к каждому из циклов будут применены различные оптимизации. В общей сложности может пройти до 200 фаз оптимизаций вплоть до планирования кода и распределения регистров. На фазе планирования кода подсчитывается время исполнения (с точностью до широкой команды без учёта блокировок конвейера) каждой копии. При штатной работе компилятора, копия цикла с большим оцененным временем исполнения удаляется из кода. Однако для проведения инструментирования необходимо оставить оба цикла, чтобы, когда будет известно точное число итераций вычислить и сравнить время исполнения. Также, необходимо учесть, что оптимизация раскрутки цикла (unroll) может изменить фактическое число итераций.

Важными параметрами, которые необходимо получить от компилятора во время проведения оптимизации overlap являются:

- времена исполнения предцикла, постцикла и тела цикла (для копии, к которой не применилась оптимизация) в тактах без учёта блокировок;
- II (initiation interval) – для копии, к которой применилась оптимизация расстояние

между соседними итерациями (с точностью до широкой команды);

- OVL – для копии, к которой применялась оптимизация, количество одновременно исполняемых итераций.

На основе проведенной подготовки можно вычислить время исполнения обеих копий цикла с точностью до широкой команды без учёта блокировок. Значения $T_{exec(overlapped)}$ и T_{exec} – это времена исполнений цикла, к которым применялась оптимизация *overlap* и не применялась соответственно.

$$T_{exec(overlapped)} = T_{pre} + T_{post} + II \times (N + OVL - I) \quad (1)$$

$$T_{exec} = T'_{pre} + T'_{post} + T_{loop} \times N' \quad (2)$$

где N – среднее число итераций цикла, которое может различаться для копий цикла из-за разного фактора раскрутки.

В формуле (1) полезным временем исполнения считается значение $II \times N$. Остальные члены будут считаться накладными расходами. Видно, что чем меньше N , тем большую роль играют накладные расходы. Эмпирически установлено, что при $N < 6$ применять оптимизацию, как правило, невыгодно.

Инструментирование оптимизации *overlap* производится с целью узнать среднее число итераций. Затем происходит сравнение среднего времени исполнения копии цикла, к которой применялась оптимизация, и времени исполнения копии, к которой оптимизация не применялась. Инструментирование происходит в 3 этапа:

- 1) На первом этапе от различных фаз компиляции во внутренние структуры компилятора происходит сбор информации о применении к циклам оптимизации *overlap*. Например, информацию о файле исходного кода можно получить от фазы дублирования циклов, тогда как точное время исполнения предциклов, постцикла и тела цикла можно получить только от фазы планирования кода и распределения регистров.
- 2) На втором этапе происходит создание дескрипторов каждого инструментированного цикла. В статической памяти пользовательской программы создаётся глобальный массив дескрипторов для каждой функции.
- 3) На третьем этапе в каждый цикл из функции вставляются операции увеличения значений счётчиков, расположенных в соответствующих дескрипторах циклов.

После исполнения программы выполняется анализ результатов и подготовка отчёта программисту с подсказками, как исправить обнаруженные в ходе анализа неточности в работе оптимизации. Для оптимизации *overlap* рассматривается три категории проблем:

- 1) *Циклы, к которым применялась оптимизация overlap.* Для таких циклов рассчитывается время исполнения копий, к которым применялась оптимизация, и к которым она не применялась. Время высчитывается с точностью до широкой команды без учёта блокировок. Времена работы сравниваются, и делается вывод о том, нужно было применять оптимизацию или нет. На рис. 2 приведён пример из отчёта программисту, где применение оптимизации *overlap* признано нецелесообразным с выдачей подсказки, как можно исправить ситуацию.
- 2) *Циклы, к которым не применялась оптимизация overlap.* Для таких циклов необходимо определить фактическое количество итераций. Если оно больше 6, то предсказатель профиля в компиляторе допустил ошибку, и необходимо помочь компилятору с помощью подсказки `#pragma loop count`. Пример такого дескриптора приведён на рис. 3.
- 3) *Циклы, к которым применялась оптимизация unroll с неоптимальным фактором раскрутки.* Данная оптимизация может уменьшить фактическое число итераций ниже эмпирически установленного предела. На данное поведение в инструменте

также существует правило проверки. В данном случае необходимо выставить подсказку компилятору "`#pragma unroll ( $K_{new}$ )`", где K_{new} – наибольший фактор раскрутки, при котором число итераций будет больше 6.

$$6 < \frac{N_{mean} \times K_{old}}{K_{new}}$$

где N_{mean} – среднее число итераций в цикле, K_{old} – фактор раскрутки, который предложил компилятор.

```
src: see.cpp 166-175
preloop counter: 536470178
body counter: 867999569
average iterations: 1.62
predicted iterations by compiler: 10
average loop exec time: 19.00
[----- Overlap -----]
  Tpre = 7.00, Tpost = 3.00
  П = 3 OVL = 3
  exec time: 19.00
  OVERLAP APPLIED
  Twin loop info:
    Tpre = 0.00, Tpost = 6.00, Tloop = 6.00
    average loop exec time: 12.00
  [SUGGESTION] overlapped loop is slower by more than 25%.
  Use "#pragma noswp" to disable overlap optimization.
[-----]
```

Рис. 2. Пример отчёта оптимизации *overlap* для одного цикла с большими накладными расходами на применение.

Fig. 2. Example of a loop report for *overlap* optimization with large overhead.

```
src: ComputeNonbondedBase.h 745-1894
head counter: 9360
body counter: 474630
average iterations: 50.71
predicted iterations by compiler: 1
average loop exec time: 52317813.58
[----- Overlap -----]
  Tpre = 26.00, Tpost = 0.00, Tloop = 1046355.75
  OVERLAP DIDN'T APPLY
  [SUGGESTION] because of low predicted iterations num: 1
  overlap optimization didn't apply.
  To apply overlap optimization try "#pragma loop count(N)"
  Suggested N: 51
[-----]
```

Рис. 3. Пример отчёта по оптимизации *overlap* для одного цикла с неверно предсказанным количеством итераций.

Fig. 3. Example of a loop report for *overlap* optimization with incorrect predicted loop iteration number.

3.3 Инструментирование оптимизации nesting

Оптимизация nesting позволяет вынести из тела цикла условный код, который имеет малую вероятность исполнения, в специально создаваемый охватывающий цикл. Оптимизация направлена на повышение эффективности конвейеризации цикла, так как при конвейеризации цикла с условным кодом все условные операции будут занимать ресурсы широкой команды, а nesting позволяет их вынести из тела цикла. Оптимизация также может вынести из цикла операции Load/Store, что улучшит работу последующих оптимизаций конвейеризации и позволит спланировать исполнение цикла с меньшим значением Π и большей степенью наложения. Пример работы оптимизации приведён на рис. 4.

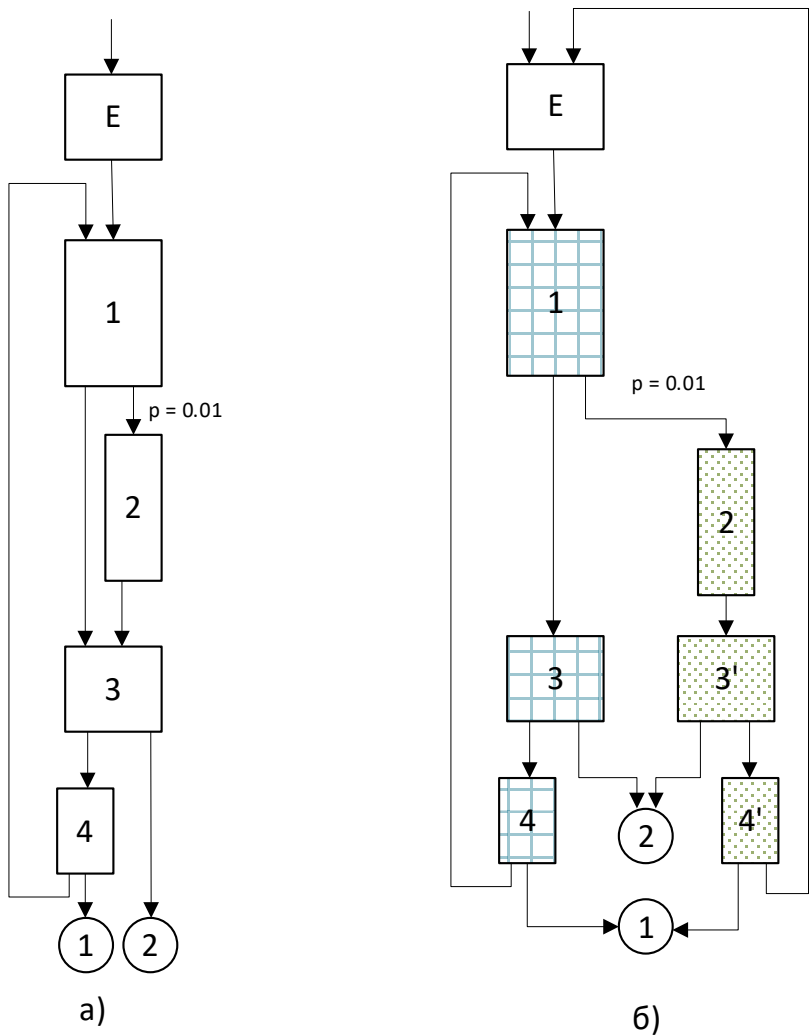


Рис. 4. а) Цикл, имеющий переход с низкой вероятностью. б) Применение оптимизации nesting.
Fig. 4. а) A loop with low branch probability. б) Example of nesting optimization applying.

На рис. 4а приведён пример графа потока управления цикла, в котором имеется условная конструкция **if**, где одна из альтернатив имеет низкую вероятность перехода. Узел *E* – входной узел для цикла (entry). Узлы 1, 2, 3, 4 образуют цикл. В кругах 1 и 2 показаны номера выходов из цикла. На рис. 4б приведён результат применения этой оптимизации. Оптимизация nesting

производит вынос альтернативы (узел 2) в охватывающий цикл (штриховка точкой), а также дублирует необходимые узлы, чтобы сохранить логику и структуру исполнения программы (узлы 3' и 4'). У внутреннего цикла (штриховка квадратами), помимо выходов 1 и 2, появляется ещё один выход по дуге с низкой вероятностью. Теперь узел 2 не относится ко внутреннему циклу и оптимизации, которые будут применяться дальше, не будут учитывать переход на данный узел, что позволит спланировать цикл с большей степенью наложения и меньшим значением Π .

Для определения частоты исполнения того или иного участка программы, оптимизация *nesting* существенно полагается на информацию о профиле исполнения программы. Поэтому при однопроходной компиляции оптимизация *nesting* практически не применяется и считается неэффективной техникой [7].

Целью инструментирования оптимизации *nesting* является нахождения отношения количества переходов по дуге в охватывающий цикл, которую создала оптимизация *nesting*, к общему числу итераций тела цикла. Так как охватывающий цикл предварительно считается кодом с низким числом итераций, это число можно проверить на практике, чтобы убедиться, что условие выхода было определено верно, а цикл не является кодом с большим количеством итераций.

На рис. 5 приведён пример инструментирования цикла. Указание $p=0.01$ приведено, чтобы выделить дугу графа потока управления, которую построила оптимизация *nesting*. Инструментирование проводится путём вставки инструкций увеличения значений счётчика итераций цикла (*cnt1*) и счётчика переходов по дуге в охватывающий цикл, который построен при проведении оптимизации *nesting* (*cnt2*).

Отметим, что здесь есть ограничение для дуг, которые ведут на блоки с гарантированным завершением. В случае, если оптимизация *nesting* посчитала переход на *catch*-блок маловероятным и вынесла его в охватывающий цикл, то построение инструкций инкрементации счётчиков на такой дуге приведут к разрыву конструкции *try-catch* и к некорректной работе программы.

После работы инструментирования и исполнения пользовательской программы, получается отчёт с подсказками, которые программист может расставить в коде для помощи компилятору. Пример такой подсказки приведён ниже на рис. 6.

4. Методика измерения производительности

Для измерения ускорения от подсказок инструмента был выбран набор тестов для измерения производительности SPEC CPU [8] 2006 и 2017 rate. Было измерено время исполнения каждого из тестов в составе набора. Затем собраны отчёты по подсказкам к компилятору с помощью предложенного инструмента. В исходном коде нами были внесены поправки в виде подсказок `#pragma {loop count, nonesting, noswp}`. Затем производилась повторная сборка тестов, замеры времени исполнения и сравнение времени исполнения с вариантом без подсказок.

Замеры происходили в базовом (*base*) режиме. Этот режим предполагает, что все тесты компилируются с одинаковыми опциями, не учитывающие особенности тестов. В данном режиме использовались опции "-O3 -fwhole".

5. Тестовый стенд

В качестве тестового стенда использовалась ЭВМ со следующими характеристиками:

- CPU: Эльбрус 8С, архитектура *elbrus-v4*, 8 ядер, частота 1.3ГГц.
- RAM: 32Гб
- OS: Эльбрус Линукс 5.4.0-2.3-nn-e8c

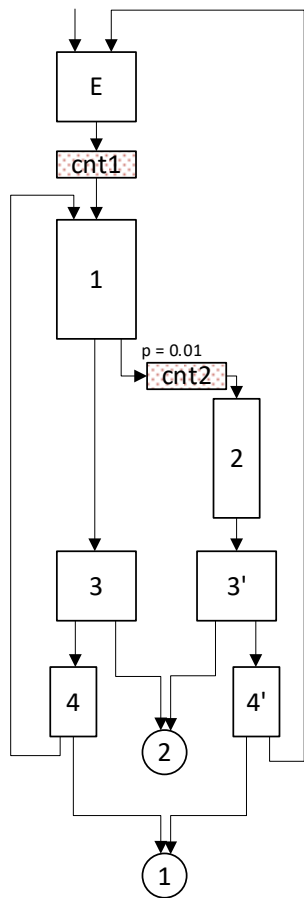


Рис. 5. Инструментирование цикла из рис. 4.
Fig. 5. Loop instrumentation from Fig. 4.

```
Src: physics_types.fppized.f90 547-551
подставлен в physics_types.fppized.f90 582
Nesting counter: 19989504
Body counter: 19989504
Ratio: 100%
[SUGGESTION]: nesting optimization seems unprofitable.
Try to use “#pragma nonesting”
```

Рис. 6. Пример отчёта для одного цикла по оптимизации nesting.
Fig. 6. Example of a loop report for nesting optimization.

6. Замеры производительности

6.1 Оптимизация overlap

На диаграммах рис. 7 и рис. 8 приведены о зафиксированном ускорении исполнения задач из наборов SPEC CPU 2006 и 2017 rate. В эти диаграммы не попали задачи, у которых либо нет подсказок, либо эффект от их расстановки не дал ни ускорения, ни замедления.



Рис. 7. Ускорение задач SPEC CPU2006
при расстановке подсказок к оптимизации overlap в режиме base.
Fig. 7. Speed up of SPEC CPU 2006 benchmark
with placement of hints to overlap optimization in base mode.

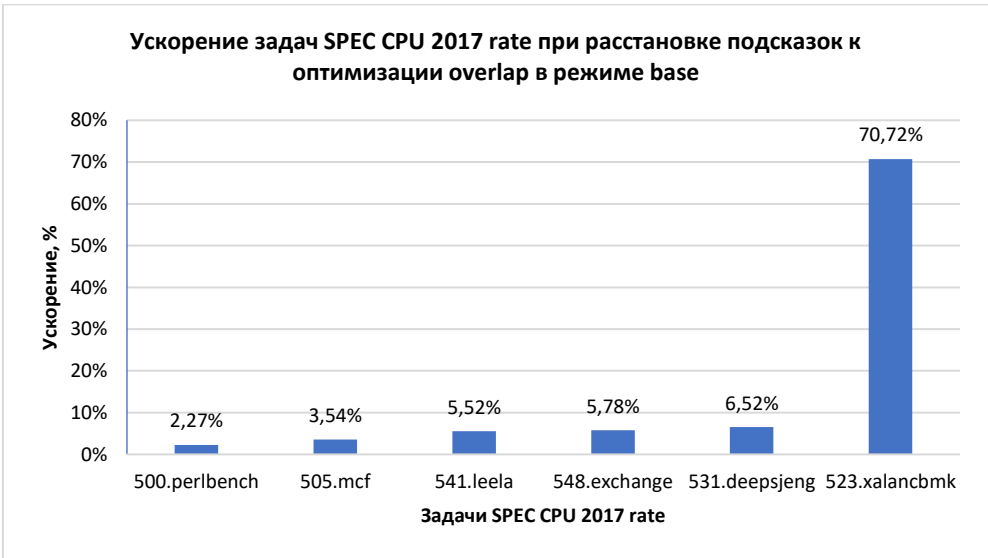


Рис. 8. Ускорение задач SPEC CPU 2017 rate
при расстановке подсказок к оптимизации overlap в режиме base.
Fig. 8. Speed up of SPEC CPU 2017 rate benchmark
with placement of hints to overlap optimization in base mode.

6.2 Оптимизация nesting

В наборе тестов SPEC CPU 2006 обнаружилась только одна задача, где проявился эффект от подсказки “`#pragma nonesting`”. Это задача *458.sjeng*, ускорение от которой составило **1.43%**. В наборе тестов SPEC CPU 2017 rate обнаружилась только две задачи: *541.leela* и *520.omnetpp*. Ускорение от расстановки подсказок составило **3.52%** и **4.71%** соответственно.

6.3 Комментарии к результатам

На примере задачи *523.xalancbmk* видно, как могут ошибаться эвристики оценки количества итераций в цикле при однопроходной компиляции. В данной задаче предсказатель профиля компилятора ошибся более, чем в половине циклов во всей задаче. В итоге, расстановка 13 подсказок вида “`#pragma loop count(N)`” позволила ускорить задачу на 70.7%.

Замедление задач *450.soplex*, *416.gamess* и *459.GemsFDTD* обусловлено сложной взаимосвязью между оптимизациями, так как одни оптимизации создают контекст для применения последующих. И даже если применение оптимизации `overlap` было избыточным, то последующие оптимизации могут всё равно ускорить цикл, тогда как без применения оптимизации `overlap` для них не появляется контекста.

Малое число обнаруженных проблем в работе оптимизации `nesting` объясняется тем, что замеры производились в базовом режиме без использования профильной информации. В разделе описания работы оптимизации `nesting` указано, что это не всегда эффективно и компилятор применяет оптимизацию очень ограниченно.

7. Выводы

В работе рассмотрены существующие методы нахождения неоптимальностей в работе оптимизаций и выявлены их недостатки. Предложен метод нахождения неоптимальностей в работе оптимизаций `overlap` и `nesting` в компиляторе LCC, путём инструментирования пользовательской программы. В результате проверки работы метода удалось ускорить часть задач из пакетов SPEC CPU 2006 и 2017 rate. Отметим, что данный метод внедрён в компилятор LCC и будет доступен пользователям начиная с версии 1.31.

Список литературы / References

- [1]. Mytkowicz T., Diwan A. et al. Evaluating the Accuracy of Java Profilers, PLDI, 2010, pp. 187-197.
- [2]. Hao X., Qingsen W. et al. Can we trust profiling results?: understanding and fixing the inaccuracy in modern profilers. Proceedings of the ACM International Conference on Supercomputing. 2019. pp. 284-295. doi:10.1145/3330345.3330371.
- [3]. Devkota S., Aschwanden P. et al. CcNav: Understanding Compiler Optimizations in Binary Code. IEEE Transactions on Visualization and Computer Graphics, 2021, pp. 667-677. doi:10.1109/TVCG.2020.3030357.
- [4]. Marques D., Duarte H. et al. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. International Conference in High Performance Computing & Simulation, 2017, pp. 898-907. doi:10.1109/HPCS.2017.150.
- [5]. Дроздов А.Ю., Степаненков А.М. Технология оптимизации циклов для архитектур с аппаратной поддержкой конвейеризации // Информационные технологии и вычислительные системы, 2004. №3, с. 52-62.
- [6]. Maslennikov D.M., Volkonsky V.Y. Compiler Method and Apparatus for Elimination of Redundant Speculative Computations from Innermost Loops, Pat. US, №6301706B1, 2001.
- [7]. Chetverina O. Alternatives of profile-guided code optimizations for one-stage compilation, Programming and Computer Software, 2016, vol. 42, no. 1, pp. 34-40. doi:10.1134/S0361768816010035.
- [8]. SPEC Benchmarks and Tools. URL: <https://spec.org/benchmarks.html#cpu> (дата обращения 02.10.2024).

Информация об авторах / Information about authors

Александр Викторович ЕРМОЛИЦКИЙ – начальник отдела разработки языкового компилятора АО «МЦСТ». Сфера научных интересов: оптимизации в компиляторах.

Alexander Viktorovich ERMOLITSKY – head of the language compiler division at MCST Design Center. Research interests: compiler optimizations.

Дмитрий Николаевич ЛЕВЧЕНКО – старший программист в АО «МЦСТ», аспирант МФТИ. Сфера научных интересов: оптимизации в компиляторах, инструментирующее профилирование, компиляторы для машинного обучения.

Dmitry Nikolaevich LEVCHENKO – software engineer at MCST Design Center and postgraduate student at MIPT. Research interests: compiler optimizations, instrumenting profiling, machine learning compilers.

Мурад Искендер-оглы НЕЙМАН-ЗАДЕ – начальник отделения разработки систем программирования в АО «МЦСТ». Его научные интересы включают методы оптимизации кода, JIT-компиляцию, профилирование и библиотеки ускоренных вычислений.

Murad Iskender-ogly NEIMAN-ZADE – head of the Department of Programming Systems at MCST Design Center. His research interests include compiler optimizations, JIT compilation, profiling and accelerated computation libraries.

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



Detection of Dead Function Calls as Source Code Defects through Static Analysis

^{1,2} V.I. Vasileva, ORCID: 0009-0008-8542-6276 <veravasileva14@ispras.ru>

¹ A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>

¹ A.E. Volkov, ORCID: 0000-0002-6043-5095 <volkov@ispras.ru>

¹ *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.*

Abstract. Detection of dead code (i.e. the code which is executed, but does not affect an observable program behavior) is commonly used by compilers as a part of optimization techniques for redundant code elimination. At the same time dead function calls might be seen as a kind of program source code defects, which may point to serious program logic faults. We describe a new detector for this kind of issues developed as a part of SVACE static defect detection tool, as well as the specific cases, which should be filtered out for practical detection of dead functions calls as program errors in contrast to their formal definition.

Keywords: static analysis; software defects; dead code; useless code; side effects; SVACE; C/C++ programming languages; Go programming language; symbolic execution; data-flow analysis; live variable analysis; interprocedural analysis.

For citation: Vasileva V.I., Borodin A.E., Volkov A.E. Detection of Dead Function Calls as Source Code Defects through Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025. pp. 65-78. DOI: 10.15514/ISPRAS-2025-37(4)-4.

Поиск мёртвых вызовов функций как дефектов исходного кода методами статического анализа

^{1,2} В.И. Васильева, ORCID: 0009-0008-8542-6276 <veravasilieva14@ispras.ru>

¹ А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>

¹ А.Е. Волков, ORCID: 0000-0002-6043-5095 <volkov@ispras.ru>

¹ Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

² Московский государственный университет имени М.В. Ломоносова,
Россия, 119991, г. Москва, Ленинские горы, д. 1.

Аннотация. Обнаружение мёртвого кода традиционно используется в компиляторах как один из видов оптимизации с целью его удаления. Мёртвые вызовы функций могут рассматриваться и как вид программных дефектов, которые могут указывать на серьёзные ошибки в реализации логики программы. В статье описывается реализация нового детектора, разработанного как часть инструмента статического анализа SVACE, а также исследуются специфические случаи мёртвых вызовов, демонстрирующие особенности задачи поиска таких ситуаций в качестве программных ошибок по сравнению с формальным определением мёртвого кода.

Ключевые слова: статический анализ; дефекты программного обеспечения; мёртвый код; бесполезный код; побочные эффекты; анализатор SVACE; языки программирования C/C++; язык программирования Go; символическое выполнение; анализ потока данных; анализ живых переменных; межпроцедурный анализ.

Для цитирования: Васильева В.И., Бородин А.Е., Волков А.Е. Поиск мёртвых вызовов функций как дефектов исходного кода методами статического анализа. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 65–78 (на английском языке). DOI: 10.15514/ISPRAS-2025-37(4)-4.

1. Introduction

In imperative programs, functions can be used both in a functional-like paradigm to calculate a return value, and like subroutines (procedures) to perform actions that will lead to some observable effects of a function call, i.e., aiming the use of their side effects.

If a function has no side effects, but is called in a way that the result of its calculations is not used, such a call is useless or has no effect. Since it wastes computational resources, a useless call itself is a sort of a source code defects, but what is more crucial, this code issue may indicate a serious fault in the program logic implementation.

Let's consider an example in Listing 1 which is based on a case we observed on an open-source projects (binutils).

As one can suppose, macro name `TO_NEXT_ID` might have misled the programmer, so the macro was used in the way like it would change the value of its argument, but in fact this code does not change it, despite the possible expectations of its author.

Dead function call is one of the possible kinds of *dead code*, and the detection of it for the purpose of its elimination is one of the common compiler optimization techniques [1, 2].

In order to avoid any terminological ambiguity, we would like to point out that use of the term *dead code* is often not quite consistent between the area of program static analysis algorithms (as well as compiler techniques) and source code defects descriptions. In this paper we use the term *dead code* to refer to the code that has no impact on the observable program behavior (which corresponds to its use for program static analysis). This meaning of dead code differs from the one for *unreachable code* and is a part of more general classes of *useless* or *redundant code*. In contrast, CWE¹ provides

¹ CWE™ – Common Weakness Enumeration

the following description in the related entry 'CWE-561: Dead Code' [3], quote: "Dead code is code that can never be executed in a running program. The surrounding code makes it impossible for a section of code to ever be executed".

In this paper we do not follow the latter definition and treat the issue described in CWE-561 as *unreachable code*.

```
1  int getNextId(int id) {
2      if (id < 0 || id >= 256)
3          return -1;
4      return id + 1;
5  }
6
7  #define TO_NEXT_ID(x) getNextId(x)
8
9  void processId(int id,
10               void (*acceptId)(int)) {
11      TO_NEXT_ID(id); // dead call to getNextId
12      acceptId(id);
13  }
```

Listing 1. Example of a dead call pointing to an erroneous program logic.

The key point of our work is to consider dead function calls as potential source code defects and we describe an implementation of a detector to search such defect issues using static analysis methods. Our experience with the detector we implemented for finding such cases demonstrates that not all the dead calls should be considered valuable as source code defects: very often the calls which are dead in formal sense and could be eliminated by the compiler during optimization, might appear due to idiomatic usage of certain programming language constructs (such as C++ templates) or specific architectural decisions (different function body implementations depending on compilation settings). Moreover, software developers might rely on the fact that these constructions will be optimized by the compiler, so an analysis that would provide warnings relevant to the programmer's perspective needs to be more specific in contrast to dead calls detection in general.

Thus, the essential feature of a detector for dead calls reported as errors is its ability to filter out the calls, which are dead formally, but have a questionable interest for the programmer (we refer to these issues as the issues, which are *true formally only*).

For this purpose, we will use an interprocedural analysis based on summaries, in which the determination of side effects occurs flow-insensitively. We will check that the return value of a function is not used using liveness analysis.

A detector for finding such erroneous issues we implemented as a new analyzer within the framework of SVACE [4, 5] static analysis tool. Our detector supports analysis of the programs written in C/C++ and Go.

2. Analysis Input Language (Internal Representation)

The low-level language we use as an internal representation for the input programs is the one of SSA-form². Its instruction kinds cover different needs, such are, for instance, some programming languages built-in intrinsics. For clarity purposes we list below only the instructions kinds, which are relevant to our paper:

² SSA – Static Single-Assignment Form: a program representation where each variable is assigned exactly once [6].

- $r := \text{alloca}()$ – allocate memory on program's call stack, write the pointer to the allocated memory to r ;
- $r := \text{load } p$ – read a value from the memory pointed by p , store this value to r ;
- $\text{store } v, p$ – write the value v to the memory pointed by p ;
- $r := v_1 \text{ op } v_2$ – apply operation op to arguments v_1 and v_2 , write its result to r , where op is one of the following: $+$, $-$, $*$, $/$, $=$, $!$, $>$, $<$, $>=$, $<=$;
- $r := \text{call } func(v_1, \dots, v_n)$ – call function $func$ and pass the values of the arguments $v_1 \dots v_n$, where $func$ is either a function name, or a variable (pointer-to-function);
- $\text{goto } L$ – unconditional jump to label L ;
- $\text{if } (v_1 \text{ op } v_2) \{ code_{true} \} \text{ else } \{ code_{false} \}$ – if $bool_expr$ yields true, then execute $code_{true}$, else execute $code_{false}$, where op is one of the operations: $=$, $!$, $>$, $<$, $>=$, $<=$, and $code_{true}$ and $code_{false}$ are instruction lists;
- $\text{return } v$ – terminate the current function execution and return the value of v to the caller's context.

Listing 2 demonstrates the representation of the program from the example in Listing 1.

```

1  def getNextId(id) {
2      t1 := alloca()
3      store id, t1
4      t2 := load t1
5      if (0 > t2) {
6          return -1
7      } else {
8          if (256 <= t2) {
9              return -1
10         } else {
11             t3 := t2 + 1
12             return t3
13         }
14     }
15 }
16
17 def processId(id, acceptId) {
18     t1 := alloca()
19     t2 := alloca()
20     store id, t1
21     store acceptId, t2
22     t3 := load t1
23     t4 := load t2
24     call getNextId(t3)
25     call t4(t3)
26     return
27 }
```

Listing 2. Example of program representation used for analysis.

3. Analysis for Dead Call Issues

3.1 Analysis Algorithm Overview

If a function execution modifies its local environment only (i.e., has no side effects), the only impact it may produce on the program execution is its return value, if it has one.

A dead function call (potentially useless from the perspective of a detector user) is a function call that has no side effects and whose result is not used further.

We will use a summary-based approach for interprocedural analysis implementation. It analyzes the functions behavior in the order of a *call graph* (CG) traversal and starts from its leaf nodes, i.e., the functions that do not call other functions. As a result of each function analysis, it creates its summary, which reflects relevant properties of the function's behavior. These summaries are used subsequently while processing calls to the analyzed functions.

During the CG traversal the engine runs side effect analysis, live variable analysis, and then the dead call issues detector for each processed function. The latter uses the results of the first two listed analyses to identify and report redundant calls.

3.2 Function Side Effects vs Function Call Side Effects

If a function has side effects, a particular call to it may have or (in contrast) may have no side effects as well. The latter can occur as a result of the conditions combination in the callee and the caller function context. We illustrate it in Listing 3.

A call to `xlog` function may have (line 12) or may have no (line 13) side effects. This fact depends on the value passed as `verbosity` argument and the actual definition of `LOG_LEVEL` macro (which is defined in this example explicitly, but may come from the compiler options in projects). In addition, this example demonstrates a case, where a true dead call issue might not be treated like a valuable defect issue. That is why in the current implementation we build our analysis to consider just the called function side effects for any call to it in a context-insensitive way.

```
1  enum { NONE=0, ERROR=1, WARNING=2, INFO=3 };
2
3  #define LOG_LEVEL WARNING
4
5  void xlog(int verbosity, int eventCode) {
6      if (verbosity <= LOG_LEVEL) {
7          printf("LOG: %d\n", eventCode);
8      }
9  }
10
11 void demo() {
12     xlog(ERROR, 123); // live call
13     xlog(INFO, 456);  // dead call
14 }
```

Listing 3. Calls with and without side effects to the same function.

3.3 Dead Call Analysis

The following cases may occur while analyzing a call to a function without side effects:

- The function does not have a return value. In this case the call is dead.
- The function has a return value, but it is not assigned to any variable. In this case the call is dead.
- The return value is assigned to an SSA variable. In this case use of live variable analysis results allows to check, if the variable is used anywhere, and if not, the call is dead.

The implemented detector identifies dead call as above, which are subject to report after additional checks to filter out issues that are irrelevant from the perspective of source code defects.

3.4 Side Effect Analysis

To determine if a function has no side effects we use a flow-insensitive analysis. A function does not have side effects, if all its instructions do not have side effects. The following instructions may have side effects:

- Function call (non-virtual). To determine if a function has side effects, the analysis uses the called function summary. If the summary does not specify the absence of side effects, the call is considered a one with side effects.
- Function call-by-pointer and virtual function calls. SVACE resolves these calls where it is possible to do [7]. In such cases, SVACE changes call instruction to the one with the resolved callee, thus the algorithm designed for direct function calls will be used then. In other cases, analysis conservatively considers these calls as the ones with side effects.
- Use of store instruction to write to memory. The instruction has a side effect, if the write occurs not to the function's local memory, i.e., memory that is not allocated by the `alloca` instruction. SVACE tracks variable aliases [8], this information is used to determine that variable is local. For cases, where it is not clear, variables are considered as non-locals.

4. Implementation

We implemented the algorithms described in the previous sections as a part of SVACE static analysis tool. SVACE supports summary-based analysis and provides a data-flow analysis engine [9], which is used to implements live variable analysis.

SVACE uses SVACE IR – an internal representation (IR) which is the same for all the supported input source code languages, the key features of SVACE IR were described in Section 2.

An important analysis feature is trace building, especially in the case of interprocedural analysis, otherwise the defects reported as its result might be unclear to users, and treated as false reports. In the case of the implemented detector, it might be not obvious, why a function does not have side effects, if it calls other functions.

The Listing 4 and the Listing 5 illustrate this and demonstrate a dead call issue reported by the implemented detector on BINUTILS (v2.22) project.

The implemented detector generates a warning as shown in Listing 6, which refers not only to the reported dead call itself at line 5, but contains information about the called functions.

```
1 static void
2 create_obj_attrs_section (void)
3 {
4     ...
5     frag_now_fix ();    // dead call
6     ...
7 }
```

Listing 4. Dead call at `as.c` from `binutils`.

```
1 #define obstack_next_free(h) ((h)->next_free)
2
3 addressT
4 frag_now_fix_octets (void)
5 {
6     if (now_seg == absolute_section)
7         return abs_section_offset;
8     return ((char *) obstack_next_free (&frchain_now->frch_obstack)
9         - frag_now->fr_literal);
10 }
11
12 addressT
13 frag_now_fix (void)
14 {
15     return frag_now_fix_octets () / OCTETS_PER_BYTE;
16 }
```

Listing 5. Interprocedural side effects absence at `frags.c` from `binutils`.

```
NO_EFFECT.CALL:
Call to function: 'frag_now_fix' has no effect at as.c:5.
  Function 'frag_now_fix' has no side effects at frags.c:16
Call to 'frag_now_fix_octets' at frags.c:15
  Function 'frag_now_fix_octets' has no side effects at frags.c:10
```

Listing 6. An example of warning report

Examining the initial detector's results on open-source projects showed us that a big amount of the detected dead function calls are just the calls to functions with empty and trivial bodies – i.e., the functions which has no instructions in the body at all or only a single return instruction that returns a constant or value of one of the formal arguments. Very often these functions look like intended stubs; their execution obviously consumes almost no computational resources and, above all, the programmers most likely expect that these calls will be eliminated by the compiler optimizations. In the case of C/C++ these functions quite often are the result of conditional preprocessing (see Section 4.2). Thus, reporting these cases as program defects is of a questionable practical interest.

Consequently, we developed additional checks in the detector to filter out the cases mentioned above in order not to report them. We devote two subsections below to the specifics of the languages supported in the current implementation of the detector. The specific we described demonstrates that though the basic detection algorithms are the same, but some language-specific constructions and widely used idioms of their use, require some additional support, that is why the current implementations does not yet cover all the languages supported by SVACE.

4.1 Go-specific Support

Go language has *channel* data type as a built-in language feature. It simplifies implementations related to *goroutines*³ communications.

Go provides *read* and *write* operations (*<-*) for channels and *select* statement, which allows a goroutine to wait for multiple channel read/write operations. For Go-specific support we extend the Exemplary Language described in Section 2 with the corresponding instructions *go_send*, *go_receive*, and *chan_select*.

Consider the representation of these instructions in an exemplary language in Listing 7 and Listing 8.

```
1 func receive(c chan int) int {
2     return <-c
3 }
4
5 func send(c chan int, value int) {
6     c <- value
7 }
8
9 func answer(in, out chan int) int {
10    select {
11    case t := <-in:
12        return t
13    case <-out:
14        return 0
15    default:
16        return -1
17    }
18 }
```

Listing 7. Example of channels use in Go programming language.

³ Goroutine – a lightweight execution thread

```
1  def receive(c) {
2      t1 := go_receive(c)
3      return t1
4  }
5
6  def send(c, value) {
7      go_send(c, value)
8      return;
9  }
10
11 def answer(in, out) {
12     t0 := chan_select(in, out)
13     t1 := extract(t0, 0)
14     if (0 != t1) {
15         if (1 == t1) {
16             t2 := extract(t0, 2)
17             return 0
18         } else {
19             return -1
20         }
21     } else {
22         t3 := extract(t0, 1)
23         return t3
24     }
25 }
```

Listing 8. IR for the example of Go program with channels use.

Any interaction with channels obviously produces a side effect, so our detector considers the related instructions of the internal representation (`go_send`, `go_receive`, and `chan_select`) as the ones with side effects.

Also Go allows to define anonymous functions inside another functions and allows the inner function to refer to local variables of its enclosing function, which forms a (*closure*). The captured variables used in these inner functions are considered as non-local.

4.2 C/C++-specific Support

C/C++ macros and certain idioms macros use introduce their own specifics to dead call issues detection.

SVACE analyzes the IR built after the source code preprocessing, therefore only a part of the actual source code might be analyzed because of conditional compilation. So, the calls detected as dead since the called function has no side effects might be not dead under other compilation conditions, when the same function produces side effects (see Listing 9). Reporting these dead calls as defect issues is undesirable. A partial solution for this problem is filtering out dummy functions with empty bodies, as we mentioned earlier in this paper.

In addition, the C++ language has templates which can be instantiated with different data types. It may happen that not all of the instantiations of a function body, which has a call to another function, but only a part of them result to a formally dead call. The analysis, which analyses the IR produced for each function body instantiation, detects only that a call in a particular instantiation is dead. However, when a call in a particular template function instantiation to another template function is dead, some other instantiation may result for the same call invocation of another instantiation of the called function, which in turn has side effects. So, a dead call in a particular instantiation does not mean a true defect issue in terms of source code in this case. Therefore, warnings about unnecessary function calls are not generated by the detector inside templates.

We illustrate the case like described above in Listing 10.

```

1 PRStatus
2 nssArena_Shutdown(void)
3 {
4     PRStatus rv = PR_SUCCESS;
5     #ifdef DEBUG
6         rv = nssPointerTracker_finalize(&arena_pointer_tracker);
7     #endif
8     return rv;
9 }
10
11 . . .
12 SECStatus
13 nss_Shutdown(void)
14 {
15     . . .
16     nssArena_Shutdown(); // questionable dead call defect
17     . . .
18 }

```

Listing 9. Questionable dead call defect because of conditional compilation in nss project.

```

1 class A {
2     static int field;
3 public:
4     int foo() {
5         return ++field;
6     }
7 };
8
9 class B {
10     static int field;
11 public:
12     int foo() {
13         return field + 1;
14     }
15 };
16
17 template <typename T>
18 void bar(T t) {
19     t.foo(); // dead call for T=B instantiation only
20 }
21
22 void demo() {
23     A a;
24     B b;
25     bar(a); // live call
26     bar(b); // dead call
27 }

```

Listing 10. Dead calls to a template function.

Method `foo()` of class `A` has side effects, since it changes variable `A::field`, in contrast to method `foo()` of class `B`. SVACE traverses both template function `bar` instantiations: `bar<A>` instantiated for `T=A` and `bar<B>` for `T=B`. The call `t.foo()` is dead inside the `bar<A>` instantiation, but not inside `bar<B>` instantiation. So a warning for the call `t.foo()` at line 19 though true for `T=B`, but false for `T=A`, thus false in general as a defect report.

5. Similar Works

Most of the static analysis tools (as well as SVACE) provide detectors for a similar from the user perspective, but different problem: unused (or ignored) return values for a specified set of functions. The list of the functions to consider is either predefined in the detector, or a tool may provide a mechanism to specify them: configuration files to describe functions behavior or source code annotations (through syntax-supported attributes, comments, etc.)

For example, KLOCWORK (proprietary) checks (SV.RVT.RETVAL_NOTTESTED issue kind [10]) for ignored return value of socket, recv and a subset of pthread-family functions.

Another tool SONARSOURCE [11] (open source) has, for example, rule RSPEC-5308 for C: “Return value of “setuid” family of functions should always be checked” and rule RSPEC-5277 for C++: “Return value of “nodiscard” functions should not be ignored”.

An open-source SPLINT [12] covers C/C++ cases, documented as: “8.4.1 Statements with No Effects” and “8.4.2 Ignored Return Values” (in “8.4 Suspicious Statements”). Their approach is based on source code attributes – ‘pure’ attribute in particular.

Note, that a tool may cover this kind of issues not for all the supported languages. For example, SONARSOURCE (already mentioned above) has a rule very close to the logic of the detector we implemented: RSPEC-2201: “Methods without side effects should not have their return values ignored”, but for C# only.

All this makes any noteworthy tools comparison quite complicated.

The closest analog for the detector described in this paper is most likely the one implemented in COVERITYSCAN. Though it is a proprietary static analysis tool and COVERITY does not provide a free access even to the list of the defect kinds, which their tool is able to detect, one of the success stories they published online [13] gives evidence of a USELESS_CALL issue detected by COVERITYSCAN on ScummVM project (file tattoo_journal.cpp), we quote a tiny part of it in Listing 11.

Commit 95884c3 in project ScummVM [14] with the message “SHERLOCK RT: Actually, clear screen instead of a useless call. CID 1308097”, which fixes this problem, is shown in Listing 12.

```
<<< CID 1308097: Incorrect expression USELESS_CALL
<<< Calling "screen->empty()" is only useful for its return value,
      which is ignored.

      screen.empty();
```

Listing 11. USELESS_CALL reported by COVERITY.

It is one more example from the open-source projects, which demonstrates that a dead code issue may indicate more serious logic faults of the analyzed program.

6. Results and Discussion

We tested the detector we have developed on the source code of open-source projects. Our measurements showed that the contribution of the new detector almost does not affect the total SVACE analysis time (it is just comparable to measurements precision).

6.1 Results on Go Projects

For the testing on Go projects we used a collection of relatively small- to medium-size projects. The detector reported 14 issues, 71% are true issues and are potential candidates to fix, while the rest is most likely of low interest for a programmer though still indicate true dead calls.

The detailed results for Go projects are displayed in Table 1. The size listed in the table includes the lines of code of analyzed dependencies (dependency libraries/projects) and for technical reasons is calculated by SVACE as a total size only.

An example of an actual dead call defect found by the detector in xgb project is shown in Listing 13. The call to Pad() method at line 12 is dead.

```
engines/sherlock/tattoo/tattoo_journal.cpp
. . .
@@ -66,7 +66,7 @@ void TattooJournal::show() {

    // Set screen to black, and set background
    screen._backBuffer1.SHblitFrom((*_journalImages)[0], Common::
        Point(0, 0));
-   screen.empty();
+   screen.clear();
    screen.setPalette(palette);

    if (_journal.empty()) {
```

Listing 12. Fix of USELESS_CALL in ScummVM project.

```
1 func Pad(n int) int {
2     return (n + 3) & ^3
3 }
4
5 func changeGContext(c *xgb.Conn, Gc Gcontext, ValueMask uint32,
6     ValueList [][]uint32) [][]byte {
7     size := xgb.Pad((8 + (4 + xgb.Pad((4
8         * xgb.PopCount(int(ValueMask)))))))
9     b := 0
10    buf := make([]byte, size)
11    ...
12    b = xgb.Pad(b) // detected dead call
13    return buf
14 }
```

Listing 13. Dead call detected in xgb project.

Table 1. Reports on Go project.

project	true formally only	true fix candidates	false	size (KLOC)
oakmound/oak	1	0	0	
tikv/tikv	0	2	0	
pingcap/tidb	1	0	0	
xgb	0	8	0	
jaeger-client-go	1	0	0	
go-redis	1	0	0	
Total:	4	10	0	>3929

6.2 Results on C/C++ Projects

The detector emitted 125 warnings on the analyzed C/C++ projects. 13 of them are true, 6 are false positives. Others though true in formal sense, but are of questionable interest for the users.

The warnings that are true formally only, relate to calls to functions, which definitions use macros and depend on conditional compilation.

False issue reports are the result of the current internal representation specifics for some built-in functions that lacks some details for them. These details are of no importance for other analyses, but are crucial in this case for ours. Thus, 6 false reports are caused by live variable analysis which does not get information to determine that the variables passed to `__builtin_alloca` are live. We plan to extend the representation used in SVACE in order to fix it.

The detailed results for C/C++ projects are shown in “Table 2”.

An example of an actual dead call defect found by the detector in `binutils` (v2.22) project is shown in Listing 14. The call to `_bfd_mips_elf_sign_extend()` at line 19 is dead.

One more example was shown in Listing 4 in Section 4.

Table 2. Reports on C/C++ projects.

project	true formally only	true fix candidates	false	size (KLOC)
binutils	8	2	0	1078.6
gnupg	3	0	0	157.5
gst-plugins-good	32	0	0	566.2
openssl	0	1	0	366.7
xorg-server	2	1	0	602.4
libxml	60	0	0	303.4
nss	2	0	0	617.6
gtk+	0	1	0	915.9
gcc	18	8	6	5390.2
Total:	125	13	6	9998.5

7. Conclusion and Future Work

As the result of our work, we developed a new detector which is a part of SVACE static analysis tool. The detector is able to find dead call issues relevant as potential source code defects. The introduced analysis has no observable impact on the overall SVACE analysis time.

We studied the results of the initial implementation versions of our detector and discovered the cases which are true in the sense of dead call definition, but are of questionable interest for software developers. Subsequently, as a significant part of the current version of the detector we implemented the algorithms to exclude the most rubbish cases from the detector output. An important part of these cases is language-specific and covers now C/C++ and Go languages. The rest of the discovered dead call cases, which are detected by the current implementation of the detector, but are of low interest as source code defect issues are the subject for the related future work and the detector improvements. In addition to that we plan to adjust the detector for other languages supported by SVACE, since their particular features and use of certain language idioms require specific support as our preliminary experiments have demonstrated to us.

```

1  bfd_vma
2  _bfd_mips_elf_sign_extend (bfd_vma value, int bits)
3  {
4      if (value & ((bfd_vma) 1 << (bits - 1)))
5          /* VALUE is negative. */
6          value |= ((bfd_vma) - 1) << bits;
7      return value;
8  }
9
10 bfd_reloc_status_type
11 _bfd_mips_elf_gprell16_with_gp (bfd *abfd, asymbol *symbol,
12     arelent *reloc_entry, asection *input_section,
13     bfd_boolean relocatable, void *data, bfd_vma gp)
14 {
15     bfd_vma relocation;
16     bfd_signed_vma val;
17
18     val = reloc_entry->addend;
19     _bfd_mips_elf_sign_extend (val, 16); // detected dead call
20
21     . . .
22
23     return bfd_reloc_ok;
24 }

```

Listing 14. Dead call detected in binutils.

References

- [1]. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: Principles, techniques, and tools (2 ed.). Addison Wesley, Boston, 2007, p. 535.
- [2]. [gcc.git] / gcc / tree-call-cdce.cc: Conditional Dead Call Elimination pass for the GNU compiler. Available at: <https://gcc.gnu.org/git/?p=gcc.git;f=gcc/tree-call-cdce.cc;hb=refs/heads/master>, accessed 19.02.2024.
- [3]. Common Weakness Enumeration (CWE™) – CWE-561: Dead Code. Available at: <https://cwe.mitre.org/data/definitions/561.html>, accessed 19.02.2024.
- [4]. Belevantsev A., Borodin A., Dudina I., Ignatiev V., Izbyshchev A., Polyakov S., Zhurikhin D. Design and Development of Svace Static Analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM), 2018, pp. 3- 9.
- [5]. Borodin A.E., Belevancev A.A. A static analysis tool Svace as a collection of analyzers with various complexity levels. Proceedings of the Institute for System Programming of the RAS, 2015, vol. 27, issue 6, pp. 111-134.
- [6]. Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K. An efficient method of computing static single assignment form. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989, pp. 25-35.
- [7]. Galustov A., Borodin A., Belevantsev A. Devirtualization for static analysis with low level intermediate representation. In 2022 Ivannikov Ispras Open Conference (ISPRAS), 2022, pp. 18-23. DOI: 10.1109/ISPRAS57371.2022.10076859.
- [8]. Borodin, A.E., Dudina, I.A. Intraprocedural Analysis Based on Symbolic Execution for Bug Detection. Programming and Computer Software, 2021, vol. 47, issue 8, pp. 858-865.
- [9]. Mulyukov, R.R., Borodin, A.E. Using unreachable code analysis in static analysis tool for finding defects in source code. Proceedings of the Institute for System Programming of the RAS, 2016, vol. 28, issue 5, pp. 145-158.
- [10]. Klocwork Documentation – C and C++ checker reference – SV.RVT.RETVAL_NOTTESTED: Ignored return value. Available at: https://help.klocwork.com/current/en-us/reference/sv.rvt.retval_nottested.htm, accessed 19.02.2024.
- [11]. SonarSource: Static Analysis Rules. Available at: <https://rules.sonarsource.com>, accessed 19.02.2024.
- [12]. Splint Manual. 8.4 Suspicious Statements. 8.4.1 Statements with No Effects. 8.4.2 Ignored Return Values. Available at: <https://splint.org/manual/manual.html#control>, accessed 19.02.2024.

- [13]. Coverity Scan – Success Stories: Sample of Defects found and fixed – ScummVM: USELESS_CAL. Available at: https://scan.coverity.com/o/oss_success_stories/86, accessed 19.02.2024.
- [14]. GitHub: ScummVM repository – commit 95884c3. Available at: <https://github.com/scummvm/scummvm/commit/95884c396b667e048af933292f40fc18da2cefd1>, accessed 19.02.2024.

Информация об авторах / Information about authors

Вера Игоревна ВАСИЛЬЕВА учится на факультете вычислительной математики и кибернетики МГУ имени М. В. Ломоносова, одновременно работая в Институте системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Vera Igorevna VASILEVA – student at the Faculty of Computational Mathematics and Cybernetics of Moscow State University (MSU), employee of Institute for System Programming of the RAS. Her research interests: compiler technologies, program source code static analysis, software defects detection.

Александр Ефимович ВОЛКОВ – старший научный сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Alexander Efimovich VOLKOV – researcher of Institute for System Programming of the RAS since 2007. Research interests: compiler technologies, program source code static analysis, software defects detection.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник Института системного программирования РАН. Сфера научных интересов: компиляторные технологии, статический анализ исходного кода программ, обнаружение ошибок программного обеспечения.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher of Institute for System Programming of the RAS since 2007. Research interests: compiler technologies, program source code static analysis, software defects detection.



HOREC: компилятор специализированных регулярных выражений для проектирования программируемой и ресурсоэффективной аппаратной архитектуры

П.Н. Советов, ORCID: 0000-0002-1039-2429 <sovetov@mirea.ru>

Федеральное государственное бюджетное образовательное учреждение высшего образования «МИРЭА — Российский технологический университет», Москва.

Аннотация. Разработка программируемых и ресурсоэффективных аппаратных ускорителей для поиска по регулярным выражениям представляет собой актуальное направление в области сетевой безопасности, где критически важны высокая пропускная способность потоковой обработки данных и устойчивость к атакам типа ReDoS (Regular Expression Denial of Service). В настоящей работе представлен компилятор HOREC, применяемый в цикле высокоуровневого проектирования программируемого аппаратного ускорителя. В HOREC используется новое расширение детерминированного конечного автомата, позволяющее компактно описывать интервальные квантификаторы с большим числом повторений, типичные для правил в системах обнаружения вторжений. Рассмотрены алгоритмы, сокращающие число переходов в командах и уменьшающие общее число команд. Представлена программная модель ускорителя на основе интерпретатора сопоставления с скомпилированным образцом и задан набор параметров для поиска в пространстве архитектурных вариантов аппаратного ускорителя, в том числе: объем доступной памяти, формат команд и режимы сопоставления символов. Экспериментальное исследование проведено для 7234 регулярных выражений, извлеченных из правил ET OPEN. Результаты оценки демонстрируют высокую ресурсоэффективность предлагаемого решения: в память объемом 64К команд удалось разместить до 7000 выражений. При этом в 60% случаев число переходов на команду не превышает 4, а наличие глобальной таблицы множеств символов для 256 часто встречающихся элементов позволяет ограничить объем памяти локальной таблицы всего 10 множествами символов на программу. Представленные результаты подтверждают применимость HOREC для аппаратной реализации поиска по РВ в задачах сетевой безопасности и показывают перспективность предложенного подхода для высокоуровневого проектирования ускорителей с низкими аппаратными затратами.

Ключевые слова: регулярные выражения; DSL-компилятор; сетевая безопасность; аппаратный ускоритель; высокоуровневое проектирование; совместное программно-аппаратное проектирование.

Для цитирования: Советов П.Н. HOREC: компилятор специализированных регулярных выражений для проектирования программируемой и ресурсоэффективной аппаратной архитектуры. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 79–96. DOI: 10.15514/ISPRAS-2025-37(4)-5.

HOREC: a Specialized Regular Expression Compiler for Designing Programmable and Resource-Efficient Hardware Architecture

P.N. Sovietov, ORCID: 0000-0002-1039-2429 <sovietov@mirea.ru>

Federal State Budgetary Educational Institution of Higher Education "MIREA - Russian Technological University", Moscow.

Abstract. The development of programmable and resource-efficient hardware accelerators for regular expression matching is an important research direction in network security, where high throughput for streaming data processing and resilience against ReDoS (Regular Expression Denial of Service) attacks are critical. This paper presents the HOREC compiler, which is used in the high-level design loop of a programmable hardware accelerator. HOREC utilizes a novel extension of a deterministic finite automaton, enabling compact representation of interval quantifiers with a large number of repetitions, typical for rules in intrusion detection systems. Algorithms are described to reduce the number of transitions in instructions and decrease the total number of instructions. The paper presents a software model of the accelerator based on an interpreter for matching compiled patterns and defines a set of parameters for architectural design space exploration of the hardware accelerator, including available memory size, instruction format, and symbols matching modes. Experimental evaluation was performed on 7234 regular expressions extracted from ET OPEN rules. The results demonstrate the high resource efficiency of the proposed solution: up to 7000 expressions were accommodated in a 64K instruction memory. Furthermore, in 60% of cases, the number of transitions per instruction does not exceed 4, and the use of a global symbol set table for 256 frequently used elements allows each program's local table to be limited to just 10 symbol sets. The presented results confirm HOREC's applicability for hardware implementation of regular expression matching in network security tasks and show the potential of the proposed approach for high-level design of low-hardware-cost accelerators.

Keywords: regular expressions; DSL compiler; network security; hardware accelerator; high-level design; hardware-software co-design.

For citation: Sovietov P.N. HOREC: a specialized regular expression compiler for designing programmable and resource-efficient hardware architecture. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 79-96 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-5.

1. Введение

Регулярные выражения (РВ) являются одним из самых популярных предметно-ориентированных языков. Поиск в строковых данных по задаваемым с помощью РВ шаблонам широко используется в таких областях, как биоинформатика, системы управления базами данных и сетевая безопасность.

Перспективным подходом для ускорения поиска по РВ в больших объемах данных и в условиях жестких временных ограничений является использование специализированных аппаратных решений.

В частности, в системах обнаружения вторжений (СОВ) к анализу сетевых пакетов предъявляется ряд требований, среди которых:

- Потоковая обработка за один просмотр и на скорости поступления данных.
- Защита от ReDoS-атак [1], направленных на значительное замедление работы системы путем подачи специально сформированных входных данных.
- Возможность оперативного обновления набора РВ.

Эти требования затрудняют использование исключительно CPU или GPU при поиске по РВ из-за непредсказуемых задержек доступа к памяти и недостаточного распараллеливания уровня задач [2]. В связи с этим, задача поиска по РВ в сетевых пакетах, зачастую, полностью или частично, делегируется специализированным аппаратным ускорителям на базе ПЛИС или СБИС.

Для представления РВ в СОВ используется автоматный подход, гарантирующий линейное время работы, в отличие от многих универсальных библиотек для РВ, использующих механизм отката. Подход на основе отката позволяет поддерживать расширенные диалекты РВ, такие как PCRE (Perl Compatible Regular Expressions), более выразительные, чем регулярные грамматики. При этом механизм отката может приводить к экспоненциальной вычислительной сложности поиска по РВ.

Трансляция РВ в ДКА позволяет в процессе поиска хранить лишь одно текущее состояние. В случае НКА требуется уже отслеживание $O(n)$ состояний, в которых одновременно может пребывать НКА в текущий момент. Несмотря на преимущество ДКА в скорости обработки, их применение сопряжено с проблемой экспоненциального роста числа состояний (до $O(2^n)$ после детерминизации НКА) и сложностью поддержки расширенных конструкций РВ. По этим причинам многие аппаратные ускорители базируются на модели НКА.

В СОВ Snort и Suricata используется специальный набор РВ, для которого, в частности, характерно наличие интервальных квантификаторов с сотнями повторений, что приводит к существенному увеличению числа состояний даже в случае НКА.

В настоящей работе представлен компилятор HOREC (Hardware-Oriented Regular Expression Compiler), предназначенный для трансляции РВ в оптимизированный код для различных архитектурных вариантов программируемого аппаратного ускорителя. Входные параметры компилятора определяют конкретный формат команд для скомпилированного РВ, а также иные, актуальные для исследуемого варианта целевой архитектуры значения. В качестве программной модели ускорителя используется универсальный интерпретатор, поддерживающий выполнение различных вариантов форматов команд. Такой цикл проектирования архитектуры аппаратного ускорителя с использованием компилятора [3-4] предполагает обратную связь со стороны специалистов в области логического синтеза и физического проектирования.

В компиляторе HOREC используется представление программ на основе нового расширения ДКА, позволяющего компактно описывать выражения с интервальными квантификаторами с большим числом повторений. Используются новые оптимизации, позволяющие сократить число команд и переходов в программах.

Полученные оценки HOREC на основе набора РВ из области сетевой безопасности демонстрируют ресурсоэффективность на уровне программной модели ускорителя. В частности, разработанные модели и алгоритмы позволяют значительно сократить объем памяти, необходимый для хранения скомпилированных РВ по сравнению с традиционными автоматными представлениями, особенно для РВ с интервальными квантификаторами.

Дальнейшее изложение имеет следующую структуру. Дается обзор архитектуры компилятора HOREC. Описываются новые модели R-НКА и R-ДКА вместе с расширенным алгоритмом детерминизации для компактного представления интервальных квантификаторов. Рассматриваются вопросы трансляции из автоматного представления в список машинных команд с использованием оптимизации для сокращения числа состояний (в HOREC это оптимизация называется Path-opt) и для сокращения числа переходов (оптимизация под названием Trans-opt). Описывается программная модель аппаратного ускорителя и алгоритм Match для интерпретации скомпилированных программ. Приводятся экспериментальные оценки компилятора HOREC. Статья завершается выводами и обсуждением перспектив дальнейших исследований.

2. Архитектура компилятора HOREC

Компилятор HOREC, архитектура которого изображена на рис. 1, принимает на вход РВ на языке подмножества PCRE, в котором не поддерживаются, в частности, предпросмотры и захваты подвыражений. Задаются параметры, описывающие варианты целевой аппаратной архитектуры. Результатом компиляции является программа в промежуточном представлении

(IR, Intermediate Representation), дополненная таблицей множеств символов. Результат компиляции используется для моделирования работы сопоставления с РВ при помощи интерпретатора. Полученная программа также может быть преобразована в двоичный формат для загрузки в память аппаратного ускорителя.

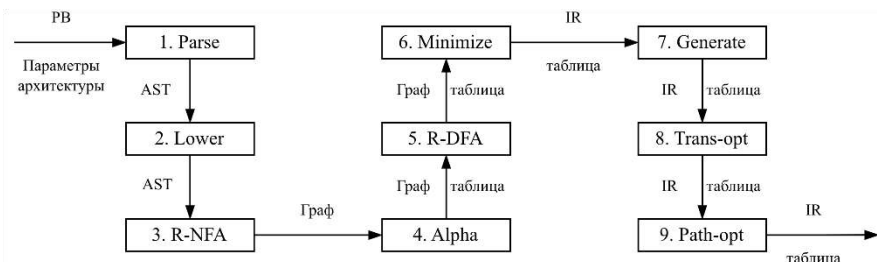


Рис. 1. Схема архитектуры компилятора HOREC.
Fig. 1. Scheme of the HOREC compiler architecture.

Компилятор написан на языке Python и имеет конвейерную архитектуру, состоящую из следующих стадий:

- 1) *Parse*. Осуществляется синтаксический разбор текста РВ с использованием библиотеки комбинаторов ресо [5]. Исходный текст РВ преобразуется в дерево абстрактного синтаксиса (AST, Abstract Syntax Tree).
- 2) *Lower*. Происходит понижение семантического уровня AST. Интервальные квантификаторы, повторяющие множество символов, транслируются в специальные R-узлы.
- 3) *R-NFA*. По входному AST строится граф НКА с поддержкой повторяемых множеств символов (R-НКА). При построении используется подход на основе частных производных Антимирова из [6], расширенный для поддержки R-узлов.
- 4) *Alpha*. На основе алгоритма Minterms из [7], осуществляется преобразование алфавита полученного графа автомата, сокращающее число переходов. Формируется таблица множеств символов.
- 5) *R-DFA*. Происходит детерминизация входного графа автомата, в процессе работы которой переходы, описывающие повторения множеств символов, пошагово раскручиваются в промежуточные состояния до достижения детерминизма.
- 6) *Minimize*. Полученный детерминированный автомат минимизируется с использованием алгоритма Бржозовского [8].
- 7) *Generate*. Граф автомата транслируется в программу в формате ИР.
- 8) *Trans-opt*. Применяется алгоритм для сокращения числа переходов в командах программы с помощью модификации таблицы множеств символов.
- 9) *Path-opt*. Применяется алгоритм для сокращения числа команд программы за счет экономного кодирования последовательностей символов в команде.

3. Построение R-НКА

Интервальные квантификаторы являются одним из основных источников сложности порождаемых НКА. Например, РВ вида $[C]\{N\}$ дает результат с не менее чем N состояниями. Для сокращения числа состояний, порождаемых интервальными квантификаторами, в компиляторе HOREC применена новая модель R-НКА.

R-НКА представляет собой частный случай обобщенного НКА, переходы которого помечены символами, являющимися РВ. Алфавит R-НКА расширен двумя дополнительными

символами, называемыми R-символами. Это символы `repset` и `repsetq` с аргументами. Аргумент C задает множество символов, а N – число повторений:

- 1) `repset(C, N)` для РВ вида $[C]\{N\}$.
- 2) `repsetq(C, N)` для РВ вида $[C]\{\emptyset, N\}$.

На стадии *Lower*, как показано на рис. 2, AST-узлы для различных типов квантификаторов сводятся к универсальному узлу `rep`, при этом частные случаи повторений множеств символов описываются с помощью R-узлов `repset` и `repsetq`, которые при дальнейшей трансляции в R-НКА становятся R-символами.

$$\begin{aligned} \text{star}(X) &= \text{rep}(X, 0, \infty) \\ \text{plus}(X) &= \text{rep}(X, 1, \infty) \\ \text{opt}(X) &= \text{rep}(X, 0, 1) \\ \text{rep}(\text{cset}(C), N, \infty) &= \text{seq}(\text{repset}(C, N), \text{rep}(\text{cset}(C), 0, \infty)) \\ \text{rep}(\text{cset}(C), N, \text{int}(M)) &= \text{seq}(\text{repset}(C, N), \text{repsetq}(C, M - N)) \end{aligned}$$

Рис. 2. Правила переписывания для квантификаторов.
Fig. 2. Rewriting rules for quantifiers.

Для построения R-НКА используется подход на основе частных производных Антимирова. Этот подход, как показано в [9], позволяет получить НКА без ϵ -переходов, в отличие от НКА Томпсона, а также дает в среднем результат с меньшим числом состояний, чем НКА Глушкова.

В оригинальном алгоритме построения НКА из [6] ключевую роль играет функция `1f` (линейная форма):

$$\text{1f}: t \rightarrow \{(c_1, t_1), \dots\}.$$

Здесь t представляет собой некоторое РВ в представлении AST, а очередная пара (c_i, t_i) описывает недетерминированный переход в состояние t_i по символу c_i .

При построении R-НКА используется функция `after`, расширяющая функцию `1f` для поддержки узлов `rep`, `repset` и `repsetq`. На рис. 3 показано определение функции `after`, а также приведены правила для вспомогательного предиката `nullable`, возвращающего 1, если его аргумент допускает пустую строку. Бинарная операция $\odot$ определена в [6].

$$\begin{aligned} \text{after}(\epsilon) &= \emptyset & \text{nullable}(\epsilon) &= 1 \\ \text{after}(\text{rep}(X, 0, 0)) &= \emptyset & \text{nullable}(\text{cset}(C)) &= 0 \\ \text{after}(\text{repset}(C, 0)) &= \emptyset & \text{nullable}(\text{rep}(X, 0, M)) &= 1 \\ \text{after}(\text{repsetq}(C, 0)) &= \emptyset & \text{nullable}(\text{rep}(X, N, M)) &= \text{nullable}(X), N \neq 0 \\ \text{after}(\text{repset}(C, N)) &= \{(t, \epsilon)\}, N \neq 0 & \text{nullable}(\text{repset}(X, 0)) &= 1 \\ \text{after}(\text{repsetq}(C, N)) &= \{(t, \epsilon)\}, N \neq 0 & \text{nullable}(\text{repset}(C, N)) &= \text{nullable}(C), N \neq 0 \\ \text{after}(\text{cset}(C)) &= \{(c, \epsilon) \mid c \in C\} & \text{nullable}(\text{repsetq}(C, N)) &= 1 \\ \text{after}(\text{rep}(X, 0, \infty)) &= \text{after}(X) \odot t & \text{nullable}(\text{seq}(X, Y)) &= \text{nullable}(X) \wedge \text{nullable}(Y) \\ \text{after}(\text{rep}(X, 0, \text{int}(M))) &= \text{after}(\text{rep}(X, 1, M)) & \text{nullable}(\text{alt}(X, Y)) &= \text{nullable}(X) \vee \text{nullable}(Y) \\ \text{after}(\text{alt}(X, Y)) &= \text{after}(X) \cup \text{after}(Y) \\ \text{after}(\text{seq}(X, Y)) &= \text{after}(X) \odot Y, \neg \text{nullable}(X) \\ \text{after}(\text{seq}(X, Y)) &= (\text{after}(X) \odot Y) \cup \text{after}(Y), \text{nullable}(X) \\ \text{after}(\text{rep}(X, N, \text{int}(M))) &= \text{after}(X) \odot \text{rep}(X, N - 1, M - 1), N \neq 0 \\ \text{after}(\text{rep}(X, N, \infty)) &= \text{after}(X) \odot \text{rep}(X, N - 1, \infty), N \neq 0 \end{aligned}$$

Рис. 3. Функция `after` для построения переходов R-НКА.
Fig. 3. *After* function for constructing R-NFA transitions.

В полученном R-НКА R-узлы `repset` и `repsetq` становятся R-символами, по которым осуществляются переходы, как показано на рис. 4а.

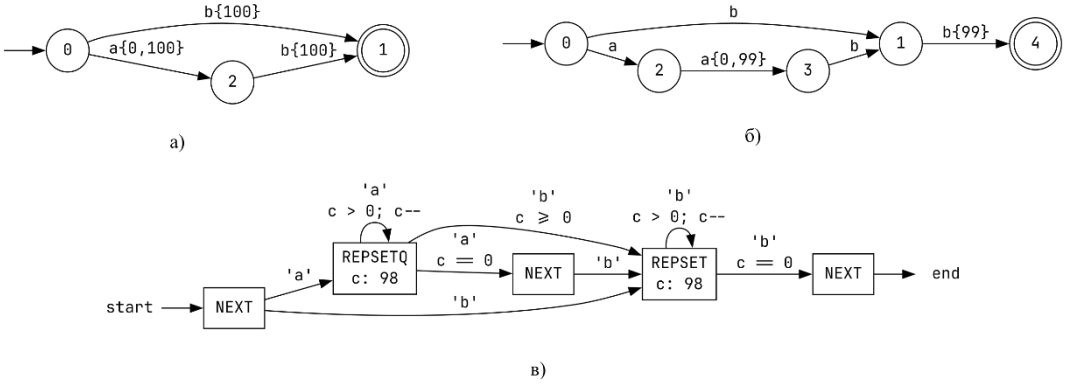


Рис. 4. Пример трансляции РВ $a\{0,100\}b\{100\}$:

а) R-НКА; б) R-ДКА; в) управляющий граф с выделенным счетчиком c .

Fig. 4. Example of translation of RE $a\{0,100\}b\{100\}$:

а) R-NFA; б) R-DFA; в) control flow graph with dedicated counter c .

Дополнительно выполняется стадия *Alpha* для сокращения числа переходов в R-НКА. Создается таблица множеств символов. Элементами нового алфавита становятся индексы множеств символов из этой таблицы. Для получения таблицы создается множество S , которое состоит из одноэлементных подмножеств, содержащих символы исходного алфавита, за исключением символов кратных переходов (для одной и той же пары состояний), объединяемых в общее подмножество. Задача создания таблицы сводится к задаче получения на основе S множества минимальной мощности S' , обладающего следующими свойствами:

- Элементы S' попарно не пересекаются.
- Каждый элемент S представим как объединение одного или нескольких элементов из S' .

Эта задача решается с помощью алгоритма Minterms из [7].

В компиляторе HOREC поддерживается режим сопоставления (match), а не поиска (search) с неявной вставкой $*$. в начало РВ. Поэтому якорь $^$ может явно размещаться только в начале РВ и наличие этого якоря в процессе компиляции игнорируется.

Для реализации якоря $\$$ используется AST-узел eof и соответствующий ему специальный символ расширенных алфавитов R-НКА и R-ДКА. Компиляция eof производится аналогичным R-узлам образом и поэтому в целях экономии объема статьи далее не рассматривается.

4. Построение R-ДКА

Для сокращения вычислительной сложности сопоставления с РВ необходимо перейти от модели R-НКА к R-ДКА. Одним из способов это сделать является предварительная полная раскрутка всех состояний, извлеченных из R-символов, за которой следует детерминизация получившегося ДКА. После этого осуществляется свертка состояний обратно в R-символы там, где это возможно. Этот способ отличается высокой ресурсоемкостью, особенно для РВ, в которых фигурируют интервальные квантификаторы с большим числом повторений.

В компиляторе HOREC для получения R-ДКА используется расширенный вариант алгоритма конструкции подмножеств, в котором раскрутка состояний для R-символов производится пошагово и по необходимости.

На рис. 5 показан псевдокод функции unroll, которая принимает на вход граф R-НКА (G) и функцию переходов (M), заданную для текущего состояния, рассматриваемого в процессе

детерминизации. Результатом `unroll` является новая функция переходов, с возможной частичной раскруткой для R-символов.

```

1: function UNROLL( $G, M$ )
2:   match  $\text{dom}(M)$  do
3:     case  $\{\text{reset}(C, N)\} \wedge N \geq \text{MIN\_N}$ 
4:       return  $M$ 
5:     case  $\{r\} \wedge r = \text{resetq}(C, N) \wedge N \geq \text{MIN\_N}$ 
6:        $M' \leftarrow \emptyset$ 
7:       for each  $e \in M(r)$  do
8:          $M' \leftarrow \text{merge}(M', \text{unroll\_moves}(G, G(e)))$ 
9:       if  $\text{set}(C) \cap \text{dom}(M') = \emptyset$  then
10:         $s \leftarrow \text{new\_state}(G, M')$ 
11:        return  $\{(r, \{s\})\}$ 
12:       else
13:        return  $\text{unroll\_moves}(G, M)$ 
14:     case  $\_$ 
15:       return  $\text{unroll\_moves}(G, M)$ 
16: function UNROLL_MOVES( $G, M$ )
17:    $M' \leftarrow \emptyset$ 
18:   for each  $(c, s) \in M$  do
19:      $M' \leftarrow \text{merge}(M', \text{unroll\_char}(G, c, s))$ 
20:   return  $M'$ 

```

Рис. 5. Псевдокод алгоритма раскрутки состояний для R-символов.

Fig. 5. Pseudocode of the state unrolling algorithm for R-symbols.

Детали реализации функции `unroll`:

- *Строка 2.* Сопоставление с областью определения M – множеством символов, по которым возможны переходы из текущего состояния.
- *Строки 3–4.* Случай, когда имеется единственный переход по символу `reset`. Этот переход является детерминированным и дальнейшей раскрутки повторяемых состояний не производится в случае, если число итераций не меньше значения архитектурного параметра `MIN_N`. Примером такого перехода является переход из состояния 1 на рис. 46.
- *Строки 5–13.* Случай, когда имеется единственный переход по R-символу `resetq`. Здесь возможен ранний выход по одному из символов, распознаваемых в множестве целевых состояний из $M(r)$.
- *Строки 6–8.* Строится функция M' для нового общего целевого состояния. При этом отдельные целевые состояния с помощью функции `unroll_moves` раскручиваются на один шаг. Функция `merge` объединяет две функции переходов, то есть два множества пар (символ, множество целевых состояний).
- *Строки 9–11.* Если множество символов из аргумента `resetq` и множество символов переходов из M' не пересекаются, то переход по `resetq`, без дальнейшей раскрутки состояний, осуществляется в созданное (функция `new_state`) общее целевое состояние, для которого заданы переходы из M' . Шаг раскрутки и создание нового состояния необходимы для того, чтобы не возникло ситуации раннего выхода по R-символу, требующей специальной обработки.
- *Строки 12–15.* Случай, когда повторяемые состояния раскручиваются на один шаг.
- *Строки 16–20.* Функция `unroll_moves` принимает на вход граф R-НКА (G) и функцию переходов (M) для текущего состояния. Результатом является новая функция переходов M' , где для каждого символа возможные повторяемые состояния раскручены на один шаг.

Псевдокод функции `unroll_char` приведен на рис. 6. Эта функция принимает на вход граф R-НКА (G), символ c и множество целевых состояний s . Результатом работы `unroll_char` является функция переходов для символа c , в которой повторяемые состояния раскручены на один шаг. Следует отметить использование рекурсии в `unroll_char` (строка 13).

Рекурсивный вызов нужен для обработки случаев, когда символы переходов целевых состояний тоже являются R-символами. Это причина появления перехода $3 \xrightarrow{b} 1$ на рис. 46. Случаи бесконечной рекурсии на нетипичных РВ вида $(x\{0,4\})^*$ выявляются на этапе компиляции.

```

1: function UNROLL_CHAR( $G, c, s$ )
2:   match  $c$  do
3:     case repset( $C, N$ )
4:       if  $N > 1$  then
5:          $s \leftarrow \{new\_state(G, \{repset(C, N - 1), s\})\}$ 
6:         return  $\{(c', s) \mid c' \in C\}$ 
7:     case repsetq( $C, N$ )
8:        $s' \leftarrow s$ 
9:       if  $N > 1$  then
10:         $s \leftarrow \{new\_state(G, \{repsetq(C, N - 1), s'\})\}$ 
11:         $M \leftarrow \{(c', s) \mid c' \in C\}$ 
12:        for each  $e \in s'$  do
13:           $M \leftarrow merge(M, unroll\_moves(G, G(e)))$ 
14:        return  $M$ 
15:     case _
16:       return  $\{(c, s)\}$ 

```

Рис. 6. Псевдокод функции `unroll_char`.

Fig. 6. Pseudocode of the `unroll_char` function.

Далее R-ДКА минимизируется с помощью алгоритма Бржозовского [8]. Этот простой алгоритм применяется без модификаций, в том числе и для переходов по R-символам.

В некоторых РВ R-символ в процессе детерминизации полностью раскручивается в отдельные состояния. Два типичных примера РВ, где это происходит:

- 1) $.^*a\{100\}$.
- 2) $a\{0,100\}a$.

Для первого примера возможно введение нового типа R-символа. Тем не менее, в общем случае поиска с префиксом $.^*$ для ДКА затруднительно избежать комбинаторного взрыва состояний. Для второго примера возможно использование правил переписывания вида

$$C\{0, N\}C = C\{1, N + 1\}.$$

5. Генератор кода

На этапе генерации кода граф R-ДКА транслируется в управляющий граф, пример которого показан на рис. 4в. Осуществляется ряд оптимизаций управляющего графа и на заключительном этапе создается список команд.

Команда включает в себя следующие поля:

- `op`. Код операции.
- `chars`. Массив кодов – индексов таблицы множеств символов.
- `jumps`. Массив адресов переходов, той же длины, что и `chars`.
- `counter`. Значение счетчика цикла для R-символов.
- `breaks`. Булев массив для команды `REPSETQ`, в котором 1 по индексу i сигнализирует о раннем выходе из цикла, переходе по коду из `chars` с тем же индексом i .
- `is_end`. Флаг допускающего состояния автомата.

Пример результата трансляции для РВ $to[^\wedge n]\{0,100\}\backslash n$:

#	op	chars	jumps	counter	breaks	is_end
0	NEXT	[3]	[1]	0	[]	0
1	NEXT	[2]	[2]	0	[]	0
2	NEXT	[0, 1, 2, 3]	[4, 3, 3, 3]	0	[]	0
3	REPSETQ	[1, 2, 3, 0]	[5, 5, 5, 4]	98	[0, 0, 0, 1]	0
4	NEXT	[]	[]	0	[]	1
5	NEXT	[0]	[4]	0	[]	0

Команда NEXT – переход в режиме обычного ДКА. Алфавит экономно закодирован, поэтому в строках 2 и 3 для множества символов $[\wedge\backslash n]$ имеется только 4 перехода. Тем не менее, кодам {1,2,3} в строке 2 соответствует общий адрес 3, а в строке 3 те же коды задают переход по общему адресу 5, то есть число переходов в некоторых командах этой программы можно сократить.

Для устранения этой избыточности в компиляторе HOREC используется алгоритм, псевдокод которого показан на рис. 7. Ключевая идея алгоритма состоит в нахождении среди элементов chars такого подмножества C , элементы которого задают переход по общему адресу j . При этом замена множества C единственным кодом n должна приводить к сокращению максимальной длины поля chars всей программы. Таким образом создается возможность перехода на формат команды меньшей длины.

```

1: function REDUCE_TRANS( $P, A$ )
2:   loop
3:      $C, n \leftarrow \text{group\_cset}(P, A)$ 
4:     if  $|C| < 2$  then
5:       return  $P, A$ 
6:      $P \leftarrow \text{replace\_cset}(P, C, n)$ 
7:      $A[n] \leftarrow \bigcup_{i \in C} A_i$ 
8: function GROUP_CSET( $P, A$ )
9:    $W \leftarrow \max(\{|p.\text{chars}| \mid p \in P\})$ 
10:   $S \leftarrow \emptyset$ 
11:   $C_o \leftarrow \emptyset$ 
12:  for each  $p \in P$  do
13:    if  $|p.\text{chars}| = W$  then
14:       $j \leftarrow \text{most\_freq}(p.\text{jumps})$ 
15:       $S \leftarrow S \cup \{p.\text{chars}[i] \mid i \in \{0, \dots, W-1\} \wedge p.\text{jumps}[i] = j\}$ 
16:    else
17:       $C_o \leftarrow C_o \cup \text{set}(p.\text{chars})$ 
18:   $C \leftarrow \bigcap_{s \in S} s$ 
19:   $U \leftarrow C - C_o$ 
20:   $n \leftarrow |A|$ 
21:  if  $|U| \neq 0$  then
22:     $n \leftarrow \min(U)$ 
23:  return  $C, n$ 

```

Рис. 7. Псевдокод алгоритма Trans-opt.
Fig. 7. Pseudocode of the Trans-opt algorithm.

Функция reduce_trans принимает на вход программу P в виде списка команд, а также таблицу множеств символов A , полученную на стадии компиляции Alpha. Результатом работы этой функции являются новая программа P' и новая таблица A' .

Детали работы функции reduce_trans:

- *Строки 3–5.* Функция reduce_trans является итеративной и выполняется до тех пор, пока очередное множество C , найденное функцией group_cset, целесообразно ($|C| > 1$) заменить кодом n .
- *Строка 6.* В функции replace_cset осуществляется замена множества C , найденного среди элементов поля chars команд программы, на код n .
- *Строка 7.* В таблицу A по индексу n заносится множество всех исходных символов, соответствующих кодам из C .

Функция group_cset:

- *Строка 9.* Значение W – максимальная длина chars среди всех команд программы,

эту длину требуется уменьшить.

- *Строки 10–11.* S и C – два множества, получаемые далее из элементов `chars` отдельных команд.
- *Строки 12–17.* Осуществляется просмотр всех команд программы.
- *Строки 13–15.* Если длина `chars` оказалась максимальной, то в S добавляется новое множество, которое состоит из тех кодов `chars`, которые указывают на один и тот же адрес перехода j . При этом j определяется, как наиболее часто встречающийся адрес в `jumps` (функция `most_freq`).
- *Строки 16–17.* В противном случае множество из `chars` добавляется в C_0 .
- *Строка 18.* Множество C состоит из тех кодов, которые фигурируют во всех элементах S .
- *Строки 19-23.* Получив C , остается выбрать новый код для n . Это может быть первый свободный индекс в A . Желательно повторно использовать один из кодов в C , при условии, что этот код более нигде не фигурирует (строка 21).

Новый вариант программы для того же РВ, но с примененным преобразованием `Trans-opt`:

#	op	chars	jumps	counter	breaks	is_end
0	NEXT	[3]	[1]	0	[]	0
1	NEXT	[2]	[2]	0	[]	0
2	NEXT	[0, 1]	[4, 3]	0	[]	0
3	REPSETQ	[0, 1]	[4, 5]	98	[1, 0]	0
4	NEXT	[]	[]	0	[]	1
5	NEXT	[0]	[4]	0	[]	0

В результате максимальное число переходов в командах сократилось с 4 до 2.

В рассматриваемом примере команды с адресами 0 и 1 задают последовательность сопоставления со строкой `T0`. В этой связи полезно ввести дополнительное преобразование `Path-opt`, в результате которого подобные пути сворачиваются в новую команду `PATH`, работающую сходным образом с `REPSET`:

#	op	chars	jumps	counter	breaks	is_end
0	PATH	[2, 3]	[1]	1	[]	0
1	NEXT	[0, 1]	[4, 2]	0	[]	0
2	REPSETQ	[0, 1]	[4, 3]	98	[1, 0]	0
3	NEXT	[0]	[4]	0	[]	0
4	NEXT	[]	[]	0	[]	1

В `PATH` коды перечислены в обратном порядке для использования при индексации счетчика цикла. Программа теперь содержит на одну команду меньше.

Для сокращения числа переходов в командах возможна также традиционная оптимизация ветвлений, при которой переход по адресу следующей команды (`pc + 1`) явно не указывается.

6. Программная модель аппаратного ускорителя

Программное моделирование сопоставления с РВ осуществляется с помощью интерпретатора `match`, псевдокод которого показан на рис. 8. На вход функции `match` подается скомпилированная программа P в формате `IR`, таблица множеств символов A и текст T . Функция возвращает булево значение результата сопоставления. В реализации `match` используются легковесные операции, допускающие непосредственную аппаратную реализацию.

<pre> 1: function MATCH(P, A, T) 2: $pos, c, pc, in_loop \leftarrow 0, 0, 0, false$ 3: loop 4: $p \leftarrow P[pc]$ 5: if $p.is_end$ then 6: return true 7: if $pos \geq T$ then 8: return $p.is_eof$ 9: if $p.op \in \{PATH, REPSET, REPSETQ\} \wedge$ $\neg in_loop$ then 10: $c \leftarrow p.counter$ 11: $in_loop \leftarrow true$ 12: $C \leftarrow p.chars$ 13: if $p.op = PATH$ then 14: $C \leftarrow C[c : c + 1]$ 15: if $(i := get_index(A, C, T_{pos})) = \perp$ then 16: return false </pre>	<pre> 17: match $p.op$ do 18: case NEXT 19: $pc \leftarrow p.jumps[i]$ 20: case PATH REPSET 21: if $c = 0$ then 22: $in_loop \leftarrow false$ 23: $pc \leftarrow p.jumps[0]$ 24: else 25: $c \leftarrow c - 1$ 26: case REPSETQ 27: if $c = 0 \vee p.breaks[i]$ then 28: $in_loop \leftarrow false$ 29: $pc \leftarrow p.jumps[i]$ 30: else 31: $c \leftarrow c - 1$ 32: $pos \leftarrow pos + 1$ </pre>
---	---

Рис. 8. Псевдокод функции match.
Fig. 8. Pseudocode of the match function.

Подробности работы функции match:

- *Строка 1.* Инициализация состояния интерпретатора: позиция pos в T , счетчик цикла c , счетчик команд pc и флаг режима выполнения цикла in_loop .
- *Строки 3–5.* Выборка очередной команды p и успешный выход при достижении допускающего состояния (is_end).
- *Строки 7–8.* Выход по завершению просмотра T . Этот выход является успешным, если текущая команда содержит переход по якорю $\$$ (is_eof).
- *Строки 9–11.* Подготовка нового цикла.
- *Строки 12–16.* Сопоставление входного символа T_{pos} с элементами $chars$. В случае команды PATH для сопоставления передается единственный элемент $chars$ по индексу, указанному в счетчике цикла. Функция get_index возвращает индекс i в массиве $chars$, если $T_{pos} \in A_n$ и $n = chars[i]$.
- *Строки 18–19.* Команда NEXT. Безусловный переход.
- *Строки 20–25.* Команды PATH и REPSET функционируют одинаково, поскольку коды $chars$ в случае PATH размещены в обратном порядке.
- *Строки 26–31.* Команда REPSETQ аналогична REPSET, но поддерживает также ранний выход из цикла при $breaks[i] = 1$.

Особенностью аппаратной реализации get_index является параллельная проверка входного символа на принадлежность одному из множеств, индексы которых указаны в $chars$. При этом для представления множеств в таблице A используются битовые вектора.

На рис. 9 представлены основные компоненты архитектуры аппаратного ускорителя с точки зрения разработчика компилятора. На вход ускорителя поступает поток данных, а сопоставление осуществляется с помощью параллельно работающих ядер Match, каждое из которых осуществляет сопоставление одного скомпилированного РВ. У ядра Match имеется локальная память для хранения скомпилированного РВ и хранения таблицы множеств символов. В составе ускорителя находится также глобальная память для часто используемых таблиц множеств символов. Со стороны CPU осуществляется настройка ускорителя и обновление набора скомпилированных программ, а сам ускоритель сигнализирует CPU о факте успешного сопоставления.

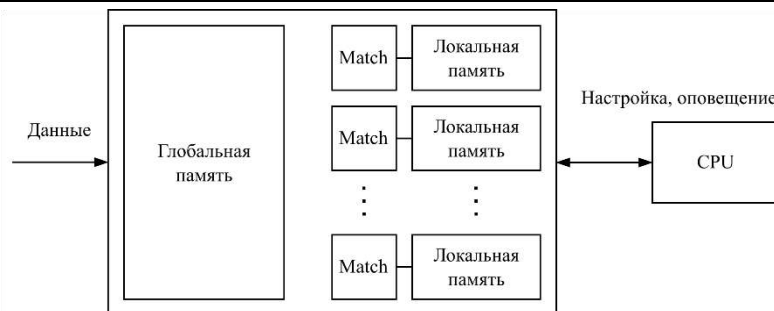


Рис. 9. Схема основных компонентов аппаратного ускорителя.
Fig. 9. Scheme of the main components of the hardware accelerator.

Представленной архитектуре аппаратного ускорителя соответствует множество вариантов его возможной микроархитектуры. При этом микроархитектурное проектирование оказывает обратное влияние на архитектуру. В этой связи важно производить с помощью компилятора поиск в пространстве архитектурных вариантов программной модели ускорителя.

Среди параметров, описывающих архитектурный вариант ускорителя, можно выделить следующие:

- Размеры локальной и глобальной памяти для программ и таблиц.
- Размеры полей команд.
- Максимальное значение счетчика циклов. Циклы, имеющие большее число итераций, могут быть разбиты на несколько команд.
- Формат команды. На уровне IR команда может иметь неограниченное число переходов, которые определяются в полях *chars*, *jumps* и *breaks*. При аппаратной реализации необходимо зафиксировать максимально допустимое число переходов. Возможен также выбор из нескольких форматов команды фиксированной длины. Например, для команды NEXT могут быть созданы ее аппаратные варианты в виде NEXT4 (до 4 переходов в команде), NEXT8 (до 8 переходов) и так далее.
- Режим последовательного сопоставления. При котором, например, две последовательные команды NEXT4 могут быть использованы вместо одной команды NEXT8. Неудачное сопоставление в первой NEXT4 при сброшенном специальном флаге *final_match* позволяет перейти к выполнению второй команды. Аналогичный подход может использоваться для команды PATH.

7. Экспериментальные оценки компилятора HOREC

В рамках эксперимента использован открытый набор правил для обнаружения угроз ET OPEN [10], из которого извлечены РВ, начинающиеся с ^. Модификаторы РВ игнорируются. Компилятор HOREC оценивался в тестовой конфигурации с на основе процессора Core i7-3770 3.40 ГГц, 32 ГБ ОЗУ и Python 3.13.1.

Общие сведения об извлеченном наборе РВ и результатах компиляции приведены в табл. 1. Следует отметить, что интервальные квантификаторы в наборе РВ, за исключением 13 сложных подвыражений, повторяют только множества символов, то есть выступают в качестве кандидатов в R-символы.

Как показано на рис. 10а, в варианте использования R-ДКА с оптимизацией Path-орт достаточно локальной памяти кода на 16 команд, чтобы поддержать 75% РВ. При этом используется в 3 раза больше программ, чем для базового ДКА.

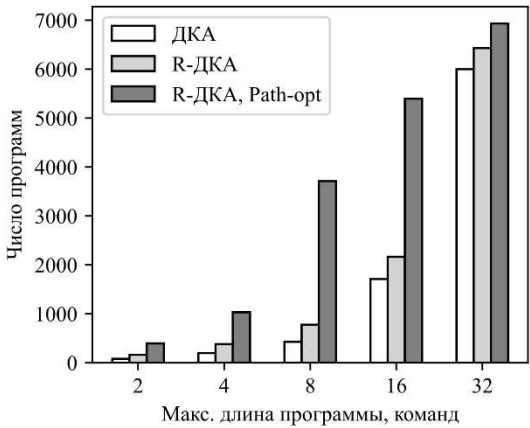
На рис. 10б выбраны РВ, содержащие интервальные квантификаторы. Здесь более заметен эффект от применения R-ДКА, чем на предыдущем графике: при ограничении объема

локальной памяти кода в 32 команды использование R-ДКА позволяет поддержать в 2 раза больше программ, чем базовый ДКА.

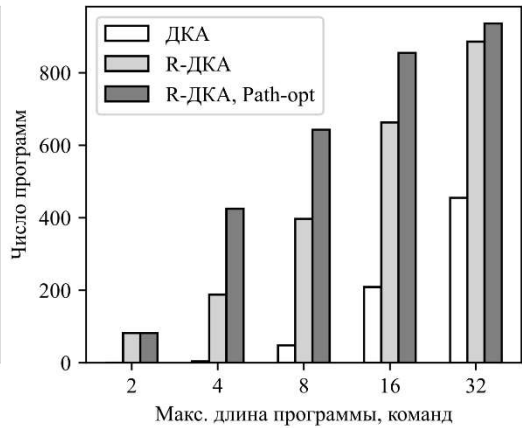
Табл. 1. Результаты компиляции РВ из набора правил ET OPEN.

Table 1. Results of RE compilation from ET OPEN ruleset.

Параметр	Значение
Всего РВ	7647
Лимит на компиляцию одного РВ, с	10
Скомпилированных РВ	7234
Скомпилированных РВ с {}	1455
Скомпилированных РВ с []{}	1442
Всего множеств в таблицах	101473
Уникальных множеств в таблицах	4220
Макс. значение счетчика цикла	999



а)



б)

Рис. 10. Распределение длин программ:

а) для общего набора РВ; б) для РВ с интервальными квантификаторами.

Fig. 10. Distribution of program sizes:

а) for the whole set of REs; б) for REs with interval quantifiers.

Вариант R-ДКА с Path-opt, как показано на рис. 11а, позволяет разместить 6728 РВ в объеме памяти кода в 65536 команд. Это в 1,7 раз больше, чем для случая базового ДКА.

В оценках объема кода необходимо учитывать не только число команд, но и ширину команды, которая определяется, в первую очередь, числом полей для возможных переходов. Из рис. 11б видно, что оптимизации Minterms и Trans-opt существенно сокращают число переходов в командах. В частности, для ограничения в 4 перехода (целесообразное ограничение ширины команды с точки зрения аппаратуры) введение Minterms позволяет разместить на ускорителе в 7 раз больше РВ, чем в базовом варианте, а совместное применение Minterms и Trans-opt — в 21 раз. Применение этих оптимизаций для формата команды с 4 полями переходов дает возможность разместить 60% РВ. Для поддержки большего числа правил может быть использован режим последовательного сопоставления.

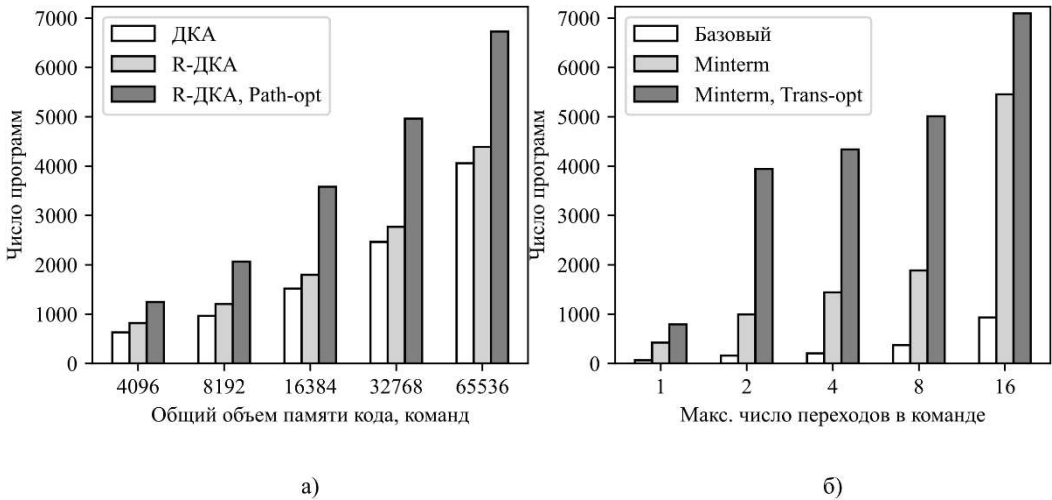


Рис. 11. Влияние оптимизаций:
а) на общий объем кода, б) на распределение числа переходов в командах.
Fig. 11. Effect of optimizations:
a) on the total code size, b) on the distribution of the number of transitions in instructions.

Распределение числа переходов (без оптимизации Path-opt) в командах на рис. 12а показывает, что наиболее часто в программах находятся команды с 1 и 2 переходами. Для этих популярных случаев могут быть спроектированы специальные варианты формата команды.

Распределение размера таблицы в программах, приведенное на рис. 12б, показывает, что наиболее часто встречаются таблицы из 16 множеств, при этом таблицы редко содержат 30 и более множеств. На основе этой информации можно определить объем локальной памяти для хранения таблицы.

В глобальной памяти возможно размещение таблицы с наиболее часто встречающимися в программах множествами. Если, как показано на рис. 13, глобальная таблица содержит 256 элементов, то размер локальной памяти таблицы может быть сокращен с 62 до 10 множеств.

В целом, такого рода статистика, полученная со стороны компилятора HOREC, может быть использована при проектировании архитектуры аппаратного ускорителя, специализированной для выбранного набора РВ.

8. Обзор существующих решений

В литературе предложен ряд моделей конечного автомата с добавленными счетчиками циклов для сокращения числа состояний, порожденных интервальными квантификаторами. В частности, в [11] предложена модель NBVA (Nondeterministic Bit Vector Automaton), которая, в отличие от настоящей работы, основана на НКА, требующем большего числа вычислительных ресурсов, чем ДКА. В [12] рассматривается модель XFA (Extended Finite Automaton) на основе ДКА, недостатком которой является низкопроизводительный алгоритм детерминизации, в процессе работы которого, в отличие от детерминизации R-НКА, производится полная раскрутка состояний с циклами. В [13] предложена модель CsA (Counting-set Automaton) с использованием сложного алгоритма детерминизации, который при получении ДКА не всегда сохраняет семантику исходного НКА, что не имеет места в случае детерминизации R-НКА. В [14] описан упрощенный вариант CsA под названием MCA (Monadic Counting Automaton), в котором, как и в R-ДКА, повторение возможно только для

классов символов. Во всех этих моделях предполагается работа со множеством счетчиков, а не с одним глобальным счетчиком, как в случае R-ДКА.

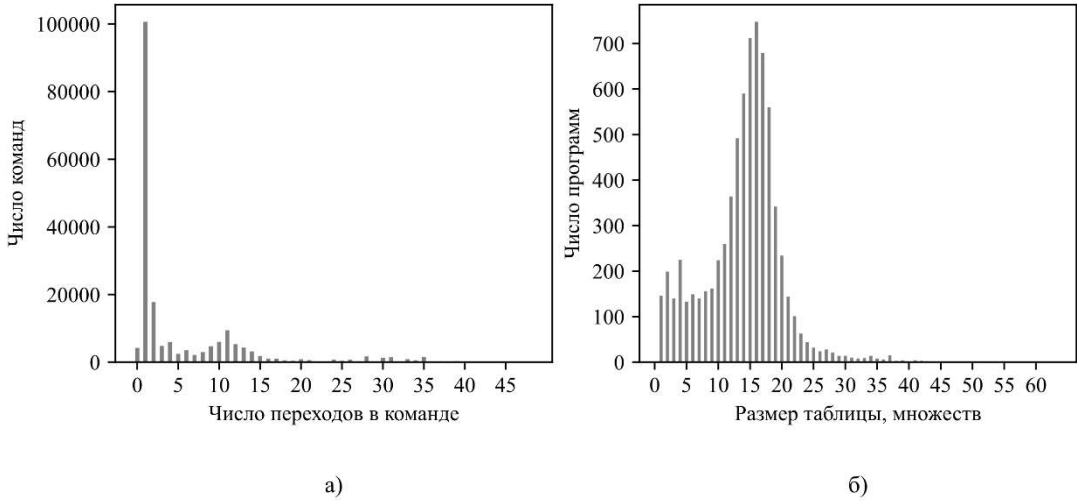


Рис. 12. Распределение:
 а) числа переходов в командах без использования Path-opt, б) размеров таблиц в программах.
 Fig. 12. Distribution of:
 а) number of transitions in instruction without using Path-opt, б) table sizes in programs.

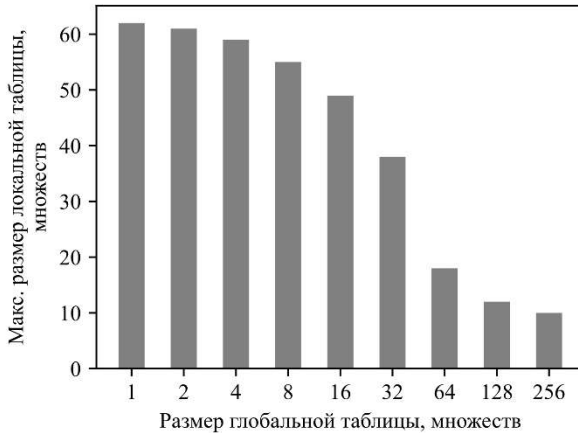


Рис. 13. Варианты размеров глобальной и локальной памяти таблиц.
 Fig. 13. Variants of global and local table memory sizes.

Многие современные аппаратные ускорители для регулярных выражений поддерживают только варианты модели НКА. При этом для решений на FPGA, таких как [15], характерен подход на основе конфигурируемости, а не программируемости, то есть использование прямой трансляции графа НКА в представление уровня RTL. Недостатком этого подхода является длительное время синтеза конфигурации FPGA, что препятствует оперативному обновлению набора регулярных выражений в задачах информационной безопасности. Существуют также автоматные процессоры, широко использующие параллелизм НКА на основе вычислений в памяти (in-memory computing). В частности, архитектура автоматного процессора BVAP (Bit Vector Automata Processor) из [16] использует модель НКА с счетчиками циклов.

Программируемым является спецпроцессор Cісего из [17], в котором для машинного представления регулярных выражений используется оперативно обновляемый программный код. Однако, в отличие от HOREC, Cісего использует набор команд для аппаратного выполнения НКА, а не ДКА.

Примером аппаратной архитектуры, использующей ДКА, является работа [18], в которой описан процессор LAP (Lightweight Automata Processor), основанный на модели A-DFA (Amortized time/bandwidth overhead DFA). Недостатком LAP является отсутствие аппаратной поддержки циклов для интервальных квантификаторов. В [19] описана архитектура XAV (Xor filter, Anchor DFA, Verification). В рамках этой архитектуры сложные РВ разбиваются на множество якорных ДКА (Anchor DFA), сформированных из регулярных подвыражений с добавленным в начало символом $\wedge$. В HOREC такие РВ также являются предпочтительными для трансляции, но они преобразуются в компактные программы, а не в таблицы ДКА.

9. Заключение

В настоящей статье рассмотрен компилятор HOREC, используемый в цикле проектирования архитектуры программируемого аппаратного ускорителя поиска по РВ. Описана архитектура компилятора и новые модели R-НКА и R-ДКА, позволяющие существенно сократить число состояний автоматов при наличии интервальных квантификаторов. Рассмотрены вопросы оптимизирующей генерации кода с использованием, в частности, нового алгоритма для сокращения числа переходов в командах. Описан алгоритм интерпретации скомпилированных программ и представлен набор архитектурных параметров для проектирования аппаратного ускорителя. Приведены результаты экспериментальной оценки разработанного компилятора, подтверждающие ресурсоэффективность на уровне программной модели ускорителя.

В дальнейшем планируется расширение модели R-ДКА для обработки более сложных случаев сворачивания повторяемых состояний в R-символы, а также для поддержки дополнительных конструкций РВ. Интерес представляет возможность объединения нескольких автоматов и поддержка вызовов процедур в скомпилированных программах. Наконец, перспективы имеет гибридная архитектура аппаратного ускорителя, часть процессорных элементов которого поддерживает выполнение на основе НКА, для сложных случаев РВ.

Список литературы / References

- [1]. Turoňová L., Holík L., Homoliak I., Lengál O., Veanes M., Vojnar T. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers / 31st USENIX Security Symposium (USENIX Security 22). — Boston, MA: USENIX Association, 2022. — С. 4165–4182.
- [2]. Carloni F., Conficconi D., Moschetto I., Santambrogio M. D. YARB: a Methodology to Characterize Regular Expression Matching on Heterogeneous Systems / 2023 IEEE International Symposium on Circuits and Systems (ISCAS). — IEEE, 2023. — С. 1–5. — DOI: 10.1109/iscas46773.2023.10181547.
- [3]. Sovietov P. N. Development of DSL Compilers for Specialized Processors // Programming and Computer Software. — Pleiades Publishing Ltd, 2021. — Т. 47, вып. 7. — С. 541–554. — DOI: 10.1134/s0361768821070082.
- [4]. Sovietov P. N. Algorithms for Improving the Automatically Synthesized Instruction Set of an Extensible Processor // Programmaya Ingeneria. — New Technologies Publishing House, 2023. — Т. 14, вып. 5. — С. 225–231. — DOI: 10.17587/prin.14.225-231.
- [5]. Tiny parser combinators library written in Python, Available at: <https://github.com/true-grue/peco>, accessed 02.08.2025.
- [6]. Antimirov V. Partial derivatives of regular expressions and finite automaton constructions // Theoretical Computer Science. — Elsevier BV, 1996. — Т. 155, вып. 2. — С. 291–319. — DOI: 10.1016/0304-3975(95)00182-4.

- [7]. D’Antoni L., Veanes M. Minimization of symbolic automata / Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — ACM, 2014. — DOI: 10.1145/2535838.2535849.
- [8]. Holzer M., Jakobi S. Brzozowski’s Minimization Algorithm—More Robust than Expected // Implementation and Application of Automata. — Springer Berlin Heidelberg, 2013. — С. 181–192. — DOI: 10.1007/978-3-642-39274-0_17.
- [9]. Sulzmann M., Lu K. Z. M. Regular expression sub-matching using partial derivatives / Proceedings of the 14th symposium on Principles and practice of declarative programming. — ACM, 2012. — С. 79–90. — DOI: 10.1145/2370776.2370788.
- [10]. Proofpoint Emerging Threats Rules, Available at: <https://rules.emergingthreats.net/>, accessed 02.08.2025.
- [11]. Le Glaunec A., Kong L., Mamouras K. Regular Expression Matching using Bit Vector Automata // Proceedings of the ACM on Programming Languages. — Association for Computing Machinery (ACM), 2023. — Т. 7, вып. OOPSLA1. — С. 492–521. — DOI: 10.1145/3586044.
- [12]. Smith R., Estan C., Jha S. XFA: Faster Signature Matching with Extended Automata / 2008 IEEE Symposium on Security and Privacy (sp 2008). — IEEE, 2008. — С. 187–201. — DOI: 10.1109/sp.2008.14.
- [13]. Turoňová L., Holík L., Lengál O., Saarikivi O., Veanes M., Vojnar T. Regex matching with counting-set automata // Proceedings of the ACM on Programming Languages. — Association for Computing Machinery (ACM), 2020. — Т. 4, вып. OOPSLA. — С. 1–30. — DOI: 10.1145/3428286.
- [14]. Holík L., Lengál O., Saarikivi O., Turoňová L., Veanes M., Vojnar T. Succinct Determinisation of Counting Automata via Sphere Construction // Programming Languages and Systems. — Springer International Publishing, 2019. — С. 468–489. — DOI: 10.1007/978-3-030-34175-6_24.
- [15]. Rahimi R., Sadredini E., Stan M., Skadron K. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs / 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). — IEEE, 2020. — С. 138–147. — DOI: 10.1109/fccm48280.2020.00027.
- [16]. Wen Z., Kong L., Le Glaunec A., Mamouras K., Yang K. BVAP: Energy and Memory Efficient Automata Processing for Regular Expressions with Bounded Repetitions / Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. — ACM, 2024. — С. 151–166. — DOI: 10.1145/3620665.3640412.
- [17]. Somaini A., Carloni F., Agosta G., Santambrogio M. D., Conficconi D. Combining MLIR Dialects with Domain-Specific Architecture for Efficient Regular Expression Matching / Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. — ACM, 2025. — С. 255–270. — DOI: 10.1145/3696443.3708916.
- [18]. Gong L., Wang C., Xia H., Chen X., Li X., Zhou X. Enabling Fast and Memory-Efficient Acceleration for Pattern Matching Workloads: The Lightweight Automata Processing Engine // IEEE Transactions on Computers. — Institute of Electrical; Electronics Engineers (IEEE), 2023. — Т. 72, вып. 4. — С. 1011–1025. — DOI: 10.1109/tc.2022.3187338.
- [19]. Zhong J., Chen S., Yu C. Xav: A High-Performance Regular Expression Matching Engine for Packet Processing. — Elsevier BV, 2024. — DOI: 10.2139/ssrn.4760269.

Информация об авторах / Information about authors

Петр Николаевич СОВЕТОВ – кандидат технических наук, доцент кафедры корпоративных информационных систем Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет». Сфера научных интересов: синтез программ, автоматизация проектирования спецпроцессоров, разработка компиляторов для спецпроцессоров.

Petr Nikolayevich SOVIETOV – Cand. Sci. (Tech.), Associate Professor of the Corporate Information Systems Department of the Institute of Information Technologies of the Institute of Information Technologies, Federal State Budgetary Educational Institution of Higher Education “MIREA - Russian Technological University”. Research interests: program synthesis, design automation of specialized processors, compiler writing for specialized processors.

DOI: 10.15514/ISPRAS-2025-37(4)-6



Extraction of Functionality from Binary Code

^{1,2} A.A. Ilina, ORCID: 0009-0002-6727-8050 <ilina@ispras.ru>

¹ Sh.F. Kurmangaleev, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>

¹ 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.

Abstract. Semantic code analysis is an important but time-consuming process used in many areas of programming. The purpose of this work is to study a method for automating the semantic analysis of binary code, which is based on dividing software into semantic kernels using partial traces of execution or subgraph extraction from call graph and highlighting their functionality.

Keywords: semantic analysis; static analysis; large language models.

For citation: Ilina A.A., Kurmangaleev Sh.F. Extraction of Functionality from Binary Code. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025, pp. 97-110. DOI: 10.15514/ISPRAS-2025-37(4)-6.

Извлечение функциональности из бинарного кода

^{1,2} А.А. Ильина, ORCID: 0009-0002-6727-8050 <ilina@ispras.ru>

¹ Ш.Ф. Курмангалеев, ORCID: 0000-0002-0558-2850 <kursh@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.

Аннотация. Семантический анализ кода – важный, но трудоемкий процесс, используемый во многих областях программирования. Целью данной работы является изучение метода автоматизации семантического анализа бинарного кода, который основан на разделении программ на семантические ядра с использованием частичных трасс выполнения или выделения подграфов графа вызовов и выделения их функциональности.

Ключевые слова: семантический анализ; статический анализ; большие языковые модели.

Для цитирования: Ильина А.А., Курмангалеев Ш.Ф. Извлечение функциональности из бинарного кода. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 97–110 (на английском языке). DOI: 10.15514/ISPRAS-2025-37(4)-6.

1. Introduction

In the modern world, software audit for certification purposes is becoming increasingly important. When conducting an audit on a large amount of code, it is essential to understand the functionality of the software under study.

However, there are often limitations:

1. The customer may not be willing to provide access to the source code for analysis due to the fact that sections of the code are proprietary.
2. Not all functionality of the code is documented.

Due to these limitations, code analysis can be time-consuming, and there is a need for automated methods to extract program functionality. The constraints define the object of analysis: it is binary code.

There are many different approaches to extracting a description of the functionality implemented in binary code. These include semantic pattern mining, semantic code clone mining and methods based on the application of neural networks. However, existing methods have limitations due to the need to keep databases of patterns or programs up to date for comparison, the need for debugging information or other facilities, and the amount of time required.

This paper will investigate a method based on static and dynamic code analysis and the use of large language models, which will allow the extraction of functionality from binary code without the need for debugging information or pattern bases. The following sections will provide a more detailed description of existing solutions, an overview of the method, comparison with analogues, experimental results, and applicability of the method to real-world problems.

2. Related Work

One method of extracting functionality from binary code involves the identification of semantic patterns. CAPA [1] is a well-known tool for identifying malicious code that relies on a similar database of patterns referred to as "rules". CAPA facilitates the identification of the semantic characteristics and functionality of binary code without necessitating full reverse-engineering. The tool employs a mapping process that translates low-level code attributes into high-level actions, such as 'makes a network connection', 'performs encryption' and 'code injection'. The primary benefit of this approach is that it enables lightweight analysis of the program. However, the approach is inherently limited by the finite size of the rule base and the laborious process of creating such rules for complex algorithms.

Another method of functionality extraction is the search for semantic code clones. By comparing the program under study with code that has been shown to have known functionality, the semantics of program fragments can be determined. Modern methods for finding semantic clones of binary code include various approaches such as symbolic execution, graph methods, use of intermediate representations (IR), as well as methods based on vector representations and machine learning. Tools such as BinDiff [2] and BinHunt [3] leverage heuristics and graph representation of code to compare its structure; however, these tools often encounter challenges when dealing with high levels of code optimization and obfuscation. In contrast, SAFE [4], which utilizes function vectorization, has exhibited both high accuracy and expeditious execution times, particularly in the context of comparing functions compiled with disparate compilers and on disparate architectures. However, even the most advanced methods, such as Gemini [5] and Asm2Vec [6], encounter limitations when dealing with heavily obfuscated code. The utilization of semantic code clone mining is warranted in scenarios where the objective is the retrieval of particular functionalities; however, the limitations that emerge in the context of a semantic pattern database also pertain to this approach.

The following discussion will proceed by way of a further examination of methodologies associated with the implementation of neural networks. The article "How Far Have We Gone in Stripped Binary

"Code Understanding Using Large Language Models" [7] reviews tools intended for semantic analysis of binary code using large language models and also creates a dataset to test the quality of these tools' performance. The article describes the sequence of steps most commonly used in the studied tools, which includes disassembling, decompiling, and neural network analysis stages. The authors note that the most common tasks in semantic analysis at the moment include function name recovery (tools for this task include NERO [8], NFRE [9], and SymLM [10]), and natural language generation for binary code segments (BinT5 [11] and HexT5 [12]). As outlined in the aforementioned article, all methods described involve a decompilation process. The employment of large language models on decompiled code has been demonstrated to yield results that surpass the accuracy of alternative methods by 10%. The utilization of language models on decompiled code demonstrates considerable promise for the purpose of learning binary code. However, the granularity of the current tasks (tools performing function-level analysis) may not be sufficient in some cases. Many binary files contain several thousand functions, and even with correctly recovered function names and generated descriptions, analyzing a binary file can take a long time. As a result, it is necessary to select semantic kernels (code areas) that implement a certain functionality in the binary code being studied.

The authors of the tool XRefer [13, 14] perform Gemini-powered cluster analysis for the binary file based on the metrics contained in the code. Like CAPA, XRefer is designed to analyze malware. It is based on the use of large language models and no longer has the constraints of a rule base. The tool can highlight any functionality. But, as mentioned, XRefer uses code metrics to analyze semantics, which may not be enough to recover functionality.

This article will delve into the process of cluster analysis which is equivalent to semantic kernels analysis, focusing on the method applied to decompiled code.

3. Method Overview

The proposed methodology in this paper is designed to determine the functionality of binary code by dividing it into sections called semantic kernels. The method is comprised of the following phases: first, sources for semantic kernels (functions in callgraph) are selected; second, semantic kernels are extracted and decompiled; third, a query is sent to a large language model for all kernels; and fourth, the functionality of the kernels is determined. The sequence of steps and the tools utilized in each step are illustrated in Fig. 1.

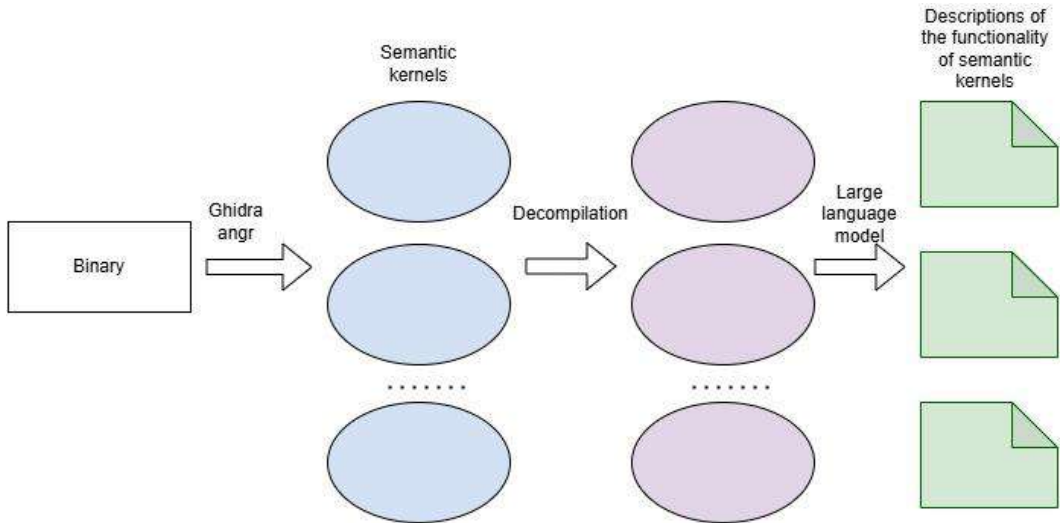


Fig. 1. General scheme of the method.

3.1 Code Division into Semantic Kernels

The goal of this stage is to reduce the number of objects required for binary code analysis by dividing the software into parts that implement specific semantics called semantic kernels. We use two types of semantic kernels.

Let's introduce their definitions:

- 1) Semantic kernel is the tree of fixed depth extracted from program's call graph, whose vertex is selected function.
- 2) Semantic kernel is the set of tracelets (partial traces of executions) that start from selected function.

In order to accomplish code division, it is necessary to select sources from which semantic kernels will be created. The selection of these sources is typically determined heuristically. Functions can be utilized as initial points for the generation of semantic kernels. In this context, semantic kernels can be considered as subgraphs of the call graph, with the vertices representing the selected sources. Fig. 2 shows semantic kernels highlighted in different colors, each with a specific functionality. For example, kernel 1 may be responsible for implementing a network protocol, kernel 2 may be responsible for reading files of a certain format, kernel 3 may be responsible for error handling, etc.

In this paper, we choose large functions with a large number of branches, strings, and comparisons as sources for semantic kernels. These functions often contain user data processing and are semantically independent from each other.

We use the Ghidra disassembler [15] to compute the metrics required for source selection.

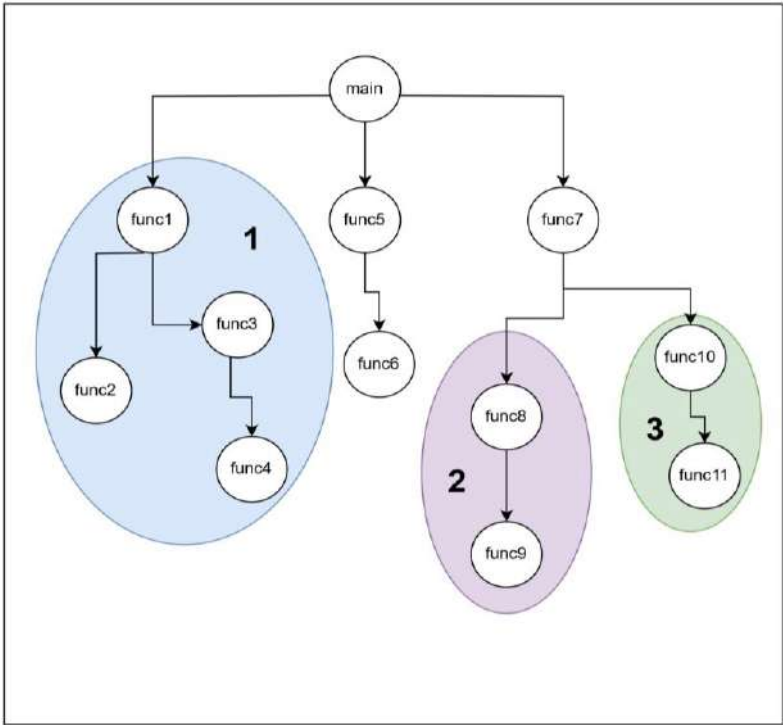


Fig. 2. Example of dividing the code into semantic kernels.

Code division into kernels is achieved in two ways as it listed before:

1. Trees of fixed depth (ranging from 2 to 3) are chosen from the call graph, whose vertices are the selected sources.

2. Angr [16] is used to generate artificial execution traces starting at the selected sources. The process of generating trails is as follows. First, the entry point of the selected function is taken to create a semantic kernel. Then instructions are taken for this entry point, which can be executed next in the control flow. The procedure is then continued for the instructions received in the previous step. Thus, the result is a set of possible execution paths starting in the function.

These techniques are employed due to limitations related to the context size of the models, as an entire call graph or large subgraphs cannot be processed by a single query.

3.2 Defining the Functionality of Semantic Kernels

The next step in the method is to define the functionality of extracted semantic kernels using a large language model. The semantic kernels obtained in the previous step are represented as binary code. In order to simplify the processing of kernels, decompilation is performed using the Ghidra. The presence of debugging information is not necessary for the proposed method to function.

A language model was chosen based on the following criteria:

1. Benchmark results. Models that successfully analyze code are of interest.
2. Local execution. This criterion ensures the security of data transmitted to the model, as it reduces the likelihood of sensitive information being leaked.
3. Output in required format. Since the tool requires an automated run and output generation, it needs the ability to convert model output into a specific format (JSON was chosen).
4. Training on data containing code samples. Given that the task involved analyzing code sections, a natural criterion was the choice of models trained on source code datasets.

According to the listed conditions, the Qwen 2.5 Coder model [17] was selected and run locally.

To determine the functionality included in the semantic kernel, algorithms are extracted from decompiled code, allowing us to understand the basic semantics of the investigated code snippet. When generating a query for the model, it was necessary to determine the level of detail in the query. Common types of queries, such as "Analyze given code snippets and identify all algorithms implemented: {CODE SNIPPET}" led to uninformative answers. A very detailed query listing a large number of algorithm types led to a longer response time, which was inefficient and unacceptable for the large number of objects being investigated. The most appropriate query is shown in Fig. 3.

3.3 Working Process

Fig. 4 presents the pseudo-code of the algorithm proposed in the paper.

4. Results

4.1 Quality Assessment of the Functionality Extraction

To evaluate the proposed method, we used TheAlgorithms repository [18], which contains implementations of various well-known algorithms in C and C++. The corresponding binary files for the algorithms were obtained from the repository. To ensure the purity of the experiment, we removed debugging information from the binary files. The algorithms in the repository were categorized. The distribution of algorithms in repository by category is presented in Fig. 5. Categories with fewer than two algorithms were excluded from consideration. For all binary files from selected categories, we performed division into semantic kernels in two ways: by extracting execution traces and by extracting subtrees from the call graph of binary. The resulting decompiled kernels were sent to a language model. The names of the algorithms in the model's answer may not exactly match the names of the algorithms whose implementations are presented in binary files. To speed up the testing process, we compare the lists of algorithms in the model output with the expected answer using st-

codesearch-distilroberta-base [19] embeddings. The testing procedure is illustrated in Fig. 6. After the results were filtered based on the threshold value, a manual validation was performed and the True Positive rate was calculated. The results for semantic kernels extracted using angr (Table 1). The best results of the method were achieved for the categories client server, hash, and searching and sorting. In most cases, the algorithms were recognized correctly. However, the minimum TP percentages (for math and misc) are related to the limitations of the angr: the generated execution traces were not long enough to fully detect the algorithms.

Prompt

Analyze the given code traces and identify all algorithms implemented in them from types: Combinatorial algorithms, Computational mathematics, Computational science, Computer science, Information theory and signal processing, Software engineering, Database algorithms, Distributed systems algorithms, Networking, Operating systems algorithms, Protocol algorithms.

If no specific algorithms are identified, respond with 'No specific algorithms identified'.

Your response should only contain the dict of algorithms by types or the 'No specific algorithms' message.

Answer must be in form {"Combinatorial algorithms": [algorithm1, ...], "Computational mathematics": [algorithm1, ...], "Computational science": [algorithm1, ...], "Computer science": [algorithm1, ...], "Information theory and signal processing": [algorithm1, ...], "Software engineering": [algorithm1, ...], "Database algorithms": [algorithm1, ...], "Distributed systems algorithms": [algorithm1, ...], "Networking": [algorithm1, ...], "Operating systems algorithms": [algorithm1, ...], "Protocol algorithms": [algorithm1, ...]}

Fig. 3. Prompt for model.

Algorithm 1 Semantic Kernels Extraction with Descriptions

```
1: Procedure EXTRACTSEMANTICKERNELS(binary, threshold, kernel_type)
2:   metrics ← calculateMetrics(binary)
3:   kernel_starts ← extractStartFunctionsByMetrics(metrics, threshold)
4:   kernels ← ∅
5:   for func in kernel_starts do
6:     if kernel_type == "angr" then
7:       kernel ← extractAngrKernel(func)
8:     else if kernel_type == "subgraph" then
9:       kernel ← extractSubgraphKernel(func)
10:    end if
11:    kernels.add(kernel)
12:  end for
13:  kernel_descriptions ← getKernelDescriptions(kernels)
14:  return kernel_descriptions
15: End Procedure
```

Fig. 4. Pseudocode of proposed method.

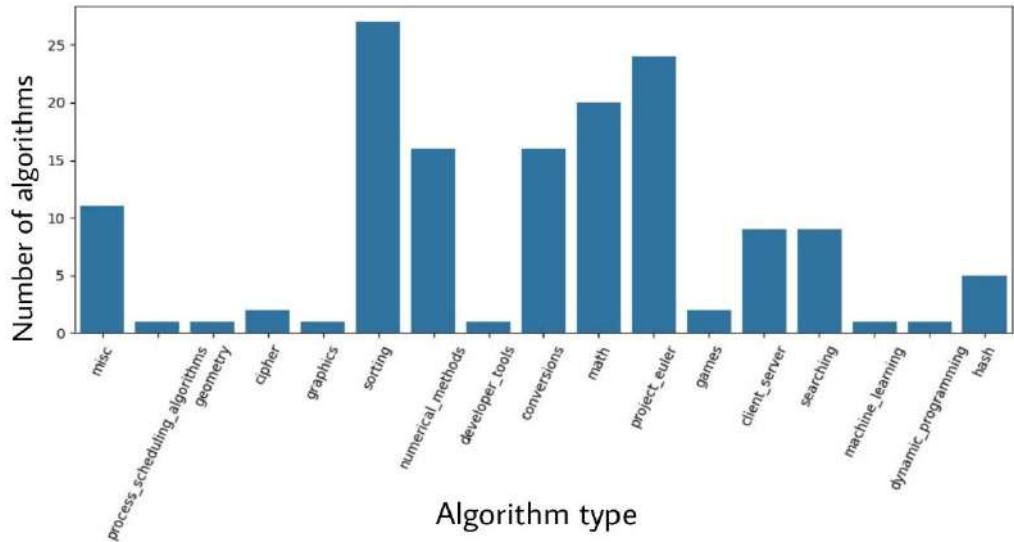


Fig. 5. The Algorithms repository structure.

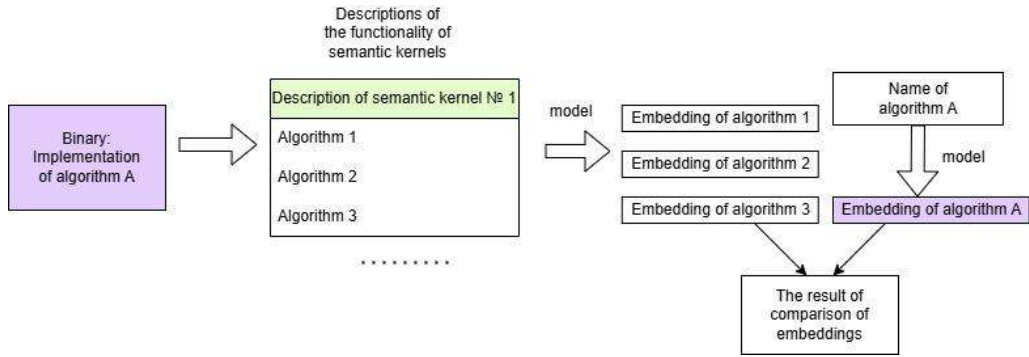


Fig. 6. Testing scheme.

Table 1. Results of using *angr* for semantic kernels extraction.

Category of algorithms	True Positive (%)
client_server	77,78
conversions	56,25
games	50
hash	80
math	45
misc	45,45
numerical_methods	50
searching	77,78
sorting	81,48

Similar testing was also conducted to extract semantic kernels using fixed-depth trees. The results are presented in Table 2. In some cases, the language model did not recognize the expected algorithms. However, in most cases, the low percentages are due to the fact that the names of the expected algorithms have a specific look (such as `remote_command_exec_udp_server`). And the response of

the model (UDP, for example) turned out to be not similar enough to the response string to be included in True positive.

Table 2. Results of using subtrees for semantic kernels extraction.

Category of algorithms	True Positive (%)
client_server	50
conversions	57,14
games	66,67
hash	83,33
math	52,17
misc	68,75
numerical_methods	44,44
searching	100
sorting	97,06

4.2 Application to Large Code Volume Analysis

As part of our research, we tested our tool on binary file of Microsoft Access 97 DBMS [20] containing 10,539 functions with no debugging information. The size of the binary is 2.9 Mb. The analysis by the Ghidra took 5 hours, the analysis using a large language model lasted an hour. This resulted in the formation of 256 clusters, each with their own functionality:

- 1) Work with databases: configuration, query processing, search, and indexes.
- 2) User interface.

Additionally, we found a cluster containing the "Magic Eight Ball" game, which was embedded as an "easter egg" by the software creators.

It was found after a series of additional prompts. The first prompt was used to extract algorithms from the software documentation. After that, the second prompt was used to compare the results of our tool with the expected algorithms in the documentation, creating a difference. This process resulted in 30 clusters being extracted. Some of these clusters contained false positives, but some also had undocumented features. To reduce the number of false positive clusters, a third prompt was used to check the generated answers. After filtering, only 5 clusters remained. These clusters were then manually checked. The remaining clusters are shown in Fig. 7.

Thus, functionality was found that was not typical for the software under study. In the future, the search for undocumented features can also be automated using queries to large language models. Fig. 8 shows the button that trigger the "Magic Eight Ball" game launch. Fig. 9 shows a function that contains a decompiled implementation of the game.

4.3 Limitations

When processing large sequences or a high volume of queries, the quality of responses may decrease. This can occur if the model loses context or is unable to process large amounts of information effectively, leading to incomplete responses. False positives may also arise as a result of hallucinations. Choosing a specialized model with a larger context size or pre-training may help reduce errors.

4.4 Comparison to Other Language Models

As mentioned earlier, the model selection criteria included the ability to run the model locally and whether the output matched the specified format in the query. We used TheAlgorithms sample to compare models that met the requirements, resulting in true positive values (see Table 3).

The testing process was performed on model CodeLlama [21] too. The output of CodeLlama for a large number of queries was not in the expected format. Qwen2.5_Coder_32B_Instruct proved to be the most suitable model for the task.

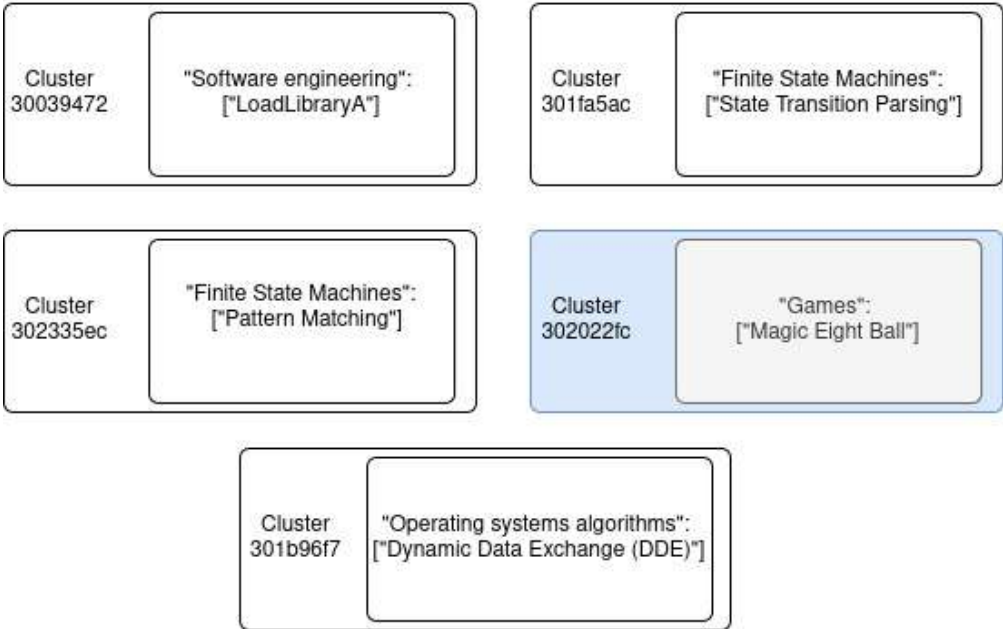


Fig. 7. Prompt sequence for undocumented features extraction.



Fig. 8. "Magic Eight Ball" in Microsoft Access 97.

4.5 Comparison to XRefer

As previously stated, XRefer employs code metrics to analyze clusters. In cases where these metrics contain sufficient information to describe the clusters (e.g., functions with numerous string constants reflecting the semantics of the code under study), cluster analysis yields satisfactory results. For

instance, the vlandhcpd [23] analysis resulted in the following set of clusters that effectively demonstrate the program semantics (see Fig. 10, DHCP Packet Processor functionality is detected). However, when the code contains few string constants but consists of a set of mathematical operations, the method is less applicable. This is demonstrated in the example of the code implementing the calculation of factorial, where XRefer did not reveal the expected functionality. As illustrated in Fig. 11, the function responsible for factorial computation does not contain string constants, leading to the failure of XRefer to recognize any functionality.



```
Decompile: FUN_302022fc - (MSACCESS.EXE)
2 void FUN_302022fc(void)
3
4 {
5     int iVar1;
6     HMODULE hModule;
7     FARPROC pFVar2;
8     HRSRC hResInfo;
9     HGLOBAL hResData;
10    LPVOID pvVar3;
11
12    iVar1 = FUN_30025860((int *)0x0);
13    hModule = LoadLibraryA("winmm.dll");
14    if ((HMODULE)0x1f < hModule) {
15        pFVar2 = GetProcAddress(hModule, "sndPlaySoundA");
16        if (pFVar2 != (FARPROC)0x0) {
17            hResInfo = FindResourceA(DAT_302c3518, (LPCSTR)0x320, &DAT_3028d3b0);
18            if (hResInfo != (HRSRC)0x0) {
19                hResData = LoadResource(DAT_302c3518, hResInfo);
20                if (hResData != (HGLOBAL)0x0) {
21                    pvVar3 = LockResource(hResData);
22                    if (pvVar3 != (LPVOID)0x0) {
23                        (*pFVar2)(pvVar3, 7);
24                    }
25                    FreeResource(hResData);
26                    FreeLibrary(hModule);
27                    FUN_300b332d((uint *)(&PTR_s_Most_likely_3028d1f8)[iVar1 % 0x12],
28                        (uint *)"The Magic Eight Ball says:", (HWND)&DAT_00000040, 0, 0, (uint *)0x0, 0);
29                }
30            }
31        }
32    }
33    return;
34 }
```

Fig. 9. Function from cluster with “Magic Eight Ball” in Ghidra.

Table 3. Model comparison.

Model	True Positive (%)
Phi3min_f16 [22]	55,56
Qwen2.5_Coder_32B_Instruct	77,78

The method proposed in the paper identified the required functionality in both vlandhcpd and factorial computation examples. This was due to analyzing the code in decompiled form. This form allows us to find code patterns that are not tied to specific metrics.

5. Conclusion

The use of large language models for semantic analysis of binary code can significantly speed up the process of understanding the functionality of a program under study. By transitioning to the level of decompiled code, large language models can help with this process. This method can be used for extracting semantic kernels from binary code, which can then be applied in software certification tasks, as well as for studying large volumes of binary code.

```
This function serves following roles in clusters:
-----
As root node in:
  • cluster.id.0005 - DHCP Packet Processor

[-] DIRECT XREFS
-----
Lookup successful                                0x52ea
Lookup of MAC %x:%x:%x:%x:%x:%x in VLAN %d      0x5283
ERROR: Unknown DHCP Option Tag: %d              0x51cd
DHCP options:                                   0x5131
Server Identifier: %d.%d.%d.%d                  0x511b
Malformed len in server identifier tag: Should be 0x5041
MSG TYPE: %s                                     0x5014
MSG TYPE: %d                                     0x4fe5
Lease: %ld                                        0x4f92
Requested IP: %d.%d.%d.%d                       0x4f45
Malformed len in requested IP tag: Should be 4 but 0x4e6b
Hostname: %s                                     0x4e31
DNS Server %d: %d.%d.%d.%d                      0x4dba
Malformed len in DNS tag: Should be a multiple of 0x4cb2
Malformed len in DNS tag: Should 4 or greater but 0x4c7d
Router %d: %d.%d.%d.%d                          0x4c36
Malformed len in router tag: Should be a multiple 0x4b2e
Malformed len in router tag: Should 4 or greater b 0x4af9
Subnet Mask: %d.%d.%d.%d                        0x4acc
Malformed len in subnet mask tag: Should be 4 but 0x49f2
%d.%d.%d.%d                                     0x4896
Optionslen: %d                                  0x4822
UNKNOWN                                         0x5006
DHCPDISCOVER                                   0x5006
DHCPOFFER                                       0x5006
DHCPREQUEST                                    0x5006
```

Fig. 10. Cluster analysis by XRefer on vlandhcpd.

```
1 int64 __fastcall func1(int a1)
2 {
3     if ( a1 )
4         return a1 * func1((unsigned int)(a1 - 1));
5     else
6         return 1LL;
7 }

XRefer:

[0x11e9] NO FUNCTION CONTEXT AVAILABLE
```

Fig. 11. Cluster analysis by XRefer on factorial computation.

References

[1]. capa: Automatically Identify Malware Capabilities. Mandiant. Google Cloud Blog. [Online]. Available at: <https://cloud.google.com/blog/topics/threat-intelligence/capa-automatically-identify-malware-capabilities>, accessed 07.04.2025.

[2]. zynamics.com – BinDiff. [Online]. Available at: <https://www.zynamics.com/bindiff.html>, accessed 07.04.2025.

[3]. D. Gao, M. K. Reiter, and D. Song, BinHunt: Automatically Finding Semantic Differences in Binary Programs in Information and Communications Security, vol. 5308, L. Chen, M. D. Ryan, and G. Wang, Eds. In Lecture Notes in Computer Science, vol. 5308. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–255. doi: 10.1007/978-3-540-88625-9_16.

- [4]. L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, SAFE: Self-Attentive Function Embeddings for Binary Similarity, Dec. 19, 2019, arXiv: arXiv:1811.05296. doi: 10.48550/arXiv.1811.05296.
- [5]. X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017, pp. 363–376. doi: 10.1145/3133956.3134018.
- [6]. S. H. H. Ding, B. C. M. Fung, and P. Charland, Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2019, pp. 472–489. doi: 10.1109/SP.2019.00003.
- [7]. X. Shang et al., How Far Have We Gone in Binary Code Understanding Using Large Language Models. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Flagstaff, AZ, USA: IEEE, Oct. 2024, pp. 1–12. doi: 10.1109/ICSME58944.2024.00012.
- [8]. Y. David, U. Alon, and E. Yahav, Neural Reverse Engineering of Stripped Binaries using Augmented Control Flow Graphs. *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–28, Nov. 2020, doi: 10.1145/3428293.
- [9]. H. Gao, S. Cheng, Y. Xue, and W. Zhang, A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Denmark: ACM, Jul. 2021, pp. 607–619. doi: 10.1145/3460319.3464804.
- [10]. X. Jin, K. Pei, J. Y. Won, and Z. Lin, SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles CA USA: ACM, Nov. 2022, pp. 1631–1645. doi: 10.1145/3548606.3560612.
- [11]. A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. van Deursen, Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. 2023, arXiv. doi: 10.48550/ARXIV.2301.01701.
- [12]. J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, HexT5: Unified Pre-Training for Stripped Binary Code Information Inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, Luxembourg: IEEE, Sep. 2023, pp. 774–786. doi: 10.1109/ASE56229.2023.00099.
- [13]. mandiant/xrefer: FLARE Team’s Binary Navigator. [Online]. Available at: <https://github.com/mandiant/xrefer>, accessed 21.02.2025.
- [14]. XRefer: The Gemini-Assisted Binary Navigator | Google Cloud Blog. [Online]. Available at: <https://cloud.google.com/blog/topics/threat-intelligence/xrefer-gemini-assisted-binary-navigator>, accessed 21.02.2025.
- [15]. Ghidra Software Reverse Engineering Framework. Available at: <https://github.com/NationalSecurityAgency/ghidra>, accessed 31.01.2025.
- [16]. Y. Shoshitaishvili et al., SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA: IEEE, May 2016, pp. 138–157. doi: 10.1109/SP.2016.17.
- [17]. QwenLM/Qwen2.5: Qwen2.5 is the large language model series developed by Qwen team, Alibaba Cloud. [Online]. Available at: <https://github.com/QwenLM/Qwen2.5>, accessed 21.02.2025.
- [18]. The Algorithms. [Online]. Available at: <https://github.com/TheAlgorithms>, accessed 21.02.2025.
- [19]. flax-sentence-embeddings/st-codesearch-distilroberta-base. Hugging Face. [Online]. Available at: <https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base>, accessed 21.02.2025.
- [20]. Microsoft Access – Wikipedia. [Online]. Available at: https://en.m.wikipedia.org/wiki/Microsoft_Access, accessed 21.02.2025.
- [21]. codellama/CodeLlama-13b-Instruct-hf. Hugging Face. [Online]. Available at: <https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf>, accessed 21.02.2025.
- [22]. microsoft/Phi-3-mini-4k-instruct. Hugging Face. [Online]. Available at: <https://huggingface.co/microsoft/Phi-3-mini-4k-instruct>, accessed 21.02.2025.
- [23]. aheck/vlandhcpd: VLAN aware DHCP server which listens on a trunk port. [Online]. Available at: <https://github.com/aheck/vlandhcpd>, accessed 21.02.2025.

Информация об авторах / Information about authors

Анна Александровна ИЛЬИНА – студентка магистратуры ВМК МГУ, лаборант ИСП РАН. Сфера научных интересов: статический анализ бинарного кода, символьное выполнение, применение больших языковых моделей.

Anna Aleksandrovna ILINA – a graduate student at the CMC MSU and a laboratory assistant at the ISP RAS. Area of her scientific interests: static binary code analysis, symbolic execution, the use of large language models.

Шамиль Фаимович КУРМАНГАЛЕЕВ – кандидат физико-математических наук, руководитель направления разработки автономных систем и технологий для создания безопасного ПО в ИСП РАН.

Shamil Faimovich KURMANGALEEV – Cand. Sci. (Phys.-Math.), head of the development of autonomous systems and technologies for creating secure software at the ISP RAS.



DOI: 10.15514/ISPRAS-2025-37(4)-7

Обзор методов миграции программных интерфейсов приложений для объектно-ориентированных языков

^{1,2} Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>

¹ Д.М. Мельник, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>

¹ Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@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.

Аннотация. Миграция приложений – это процесс переноса программного обеспечения с одной платформы или версии прикладного интерфейса на другую. В условиях быстрого развития технологий и постоянных изменений в пользовательских предпочтениях эффективная миграция интерфейсов становится необходимостью для поддержания конкурентоспособности приложений. В статье представлен обзор современных методов миграции программных интерфейсов приложений, акцентирующий внимание на важности адаптации программного обеспечения к изменяющимся условиям и требованиям пользователей. Статья также классифицирует существующие подходы к миграции, включая использование автоматизированных инструментов, методов адаптации и рефакторинга кода на объектно-ориентированных языках программирования. Рассматриваются преимущества и недостатки различных методов, таких как адаптация пользовательских интерфейсов к новым платформам, миграция на основе шаблонов и использование адаптеров для обеспечения совместимости между устаревшими и новыми интерфейсами. Обсуждаются вызовы, с которыми сталкиваются разработчики при миграции, включая проблемы семантического преобразования и необходимость учета специфики целевых платформ. Данный обзор будет полезен как исследователям, так и практикам, работающим в области разработки программного обеспечения, предоставляя знания о методах и подходах к успешной миграции программных интерфейсов приложений.

Ключевые слова: программный интерфейс приложения; миграция программного кода; миграция интерфейса API; статический анализ; трансформация программного кода; миграция библиотеки.

Для цитирования: Чуркин Я.А., Мельник Д.М., Бучацкий Р.А. Обзор методов миграции программных интерфейсов приложений для объектно-ориентированных языков. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 111–146. DOI: 10.15514/ISPRAS-2025-37(4)-7.

Survey of Application Program Interface Migration Methods for Object-Oriented Languages

^{1,2} Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>

¹ D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>

¹ R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@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.*

Abstract. Application migration is the process of moving software from one platform or API version to another. With rapid technological development and constant changes in user preferences, effective interface migration is becoming a necessity to maintain the competitiveness of applications. This article provides an overview of modern methods of application programming interface migration, focusing on the importance of adapting software to changing conditions and user requirements. The article also classifies existing approaches to migration, including the use of automated tools, adaptation and refactoring methods for code on object-oriented languages. The advantages and disadvantages of various methods, such as adapting user interfaces to new platforms, template-based migration, and using adapters to ensure compatibility between legacy and new interfaces are considered. The challenges faced by developers during migration are discussed, including semantic transformation issues and the need to take into account the specifics of target platforms. This review will be useful for both researchers and practitioners working in the field of software development, providing knowledge about methods and approaches to successful migration of application programming interfaces.

Keywords: application interface API; code migration; API migration; static analysis; code transformation; library migration.

For citation: Churkin Y.A., Melnik D.M., Buchatskiy R.A. Survey of Application Program Interface Migration Methods for object-oriented languages. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025. pp. 111-146 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-7.

1. Введение

В настоящее время различные операционные системы (ОС) быстро эволюционируют, что приводит к внедрению новых возможностей, элементов пользовательских интерфейсов и каркасов для разработки программных приложений, позволяющих использовать новейшую функциональность. Эти изменения неизбежно влекут за собой обновления в программных интерфейсах приложений (API – Application Programming Interface) [1]. В таких условиях важным становится сохранение совместимости приложений, разработанных для более старых версий операционных систем. Адаптация программных приложений к различным программным средам при изменении API, на основе которого они написаны, приобретает особую значимость.

Разработчики приложений обязаны учитывать не только изменения в API операционных систем, но и изменения в API библиотек, используемых в процессе разработки, изменения интерфейса системы сборки (например, CMake [2]) а также обновления или дополнения в языках программирования, на которых написаны эти приложения. Применение нового функционала библиотек или конструкций языка программирования может существенно повысить эффективность работы приложения. Однако изменения в API библиотеки могут привести к полной несостоятельности старого кода, написанного под предыдущие версии API.

Миграция программного кода – это процесс адаптации программного кода к обновлениям одного или нескольких используемых им программных компонентов путем его трансформации. В зависимости от сложности миграции, ее ручное осуществление может

быть достаточно долгим и трудозатратным процессом, особенно для больших кодовых баз. В связи с этим остро встает вопрос о способах автоматизации этого процесса.

Целями исследования является классификация типов миграции кода на объектно-ориентированных языках (раздел 2), а также обзор инструментов и методов автоматической и автоматизированной миграции кода (раздел 3).

Доступность программных решений и исследований, касающихся адаптации программных приложений под различные версии конкретного API, остается ограниченной. В данной работе будут рассмотрены основные из них.

Основные результаты исследования могут быть использованы разработчиками приложений для определения возможности осуществления необходимых миграций их кодовых баз автоматическим или полуавтоматическим способом и помочь с выбором инструмента для осуществления необходимых миграций.

2. Классы миграции программных интерфейсов приложений

Согласно опубликованной статистике [3] менее 40% разработчиков приложений под операционную систему Android полностью адаптировали свои приложения под версию 14 (версия API 34). Под актуальную версию Android 15 (версия API 35) адаптация приложений составляет менее 2%. Изменение версии Android и его API происходят ежегодно и политика выпуска новых версий API в основном направлена на добавление новых функциональных возможностей, улучшение безопасности и энергоэффективности работы приложений и не гарантирует обратной совместимости. Это означает, что рано или поздно приложение может перестать корректно работать или работать не так эффективно на новых версиях Android. Разработчики, которые долго не обновляют версию API своих приложений подвергают риску пользователей. Так как приложений, написанных для Android достаточно много и смена версий API происходит часто, проблема миграции Android приложений под новые версии API стоит особенно остро [4]. Согласно статистике [5], Java является наиболее популярным языком для написания Android приложений. Поэтому было принято решение отбирать примеры миграции из приложений, написанных под Android на языке Java. Эти примеры могут быть обобщены для применения к любому объектно-ориентированному языку программирования, так как являются обособленными и не привязаны к особенностям языка Java.

Отбор примеров миграции исходного кода проводился из открытых репозиторий популярных приложений и программных платформ, написанных на языке Java с открытым исходным кодом на платформе GitHub [6]. Поиск осуществлялся по записям изменений кода (commits), содержащим в заголовке или содержании информацию о переходе с одной версии Android API на другую, например, с двадцать шестой версии на двадцать восьмую и другие. В результате данного процесса было найдено порядка 60 различных примеров изменений программного кода.

После завершения отбора примеры были подвергнуты анализу с целью выявления схожих признаков изменений в программном коде. В результате все примеры были разделены на 2 основных класса миграции API:

1. Миграция с сохранением семантической эквивалентности (низкоуровневый класс миграции API).
2. Миграция без сохранения семантической эквивалентности (высокоуровневый класс миграции API).

Первый класс был дополнительно разделен на 5 подклассов из-за возможности группировки по схожести найденных примеров миграции, принадлежащих данному классу:

1. замена вызова метода или функции последовательностью инструкций;
2. замена библиотеки на схожую по функциональным возможностям;

3. переименование метода класса или функции;
4. замена сигнатуры метода класса или функции;
5. замена нестатического метода статическим и наоборот.

Примеры миграции, принадлежащие второму классу, не были дополнительно сгруппированы из-за уникальности каждого найденного примера. В следующих подразделах будет представлено более подробное описание каждого из этих классов миграции API.

2.1 Миграция с сохранением семантической эквивалентности (низкоуровневый класс миграции API)

Данный класс миграции API относится к ситуации, когда гарантируется семантическая эквивалентность кода до и после миграции. Далее будет дано подробное описание подклассов данного класса миграции.

2.1.1 Замена вызова метода/функции/части кода последовательностью инструкций

Данный подкласс миграции API описывает ситуацию, когда вызов метода класса, функции или часть кода должны быть заменены последовательностью инструкций с сохранением семантической эквивалентности.

На листинге 1 представлен пример замены вызова метода `getAllNetworkInfo` Java-класса `ConnectivityManager` на метод `getAllNetworks` того же класса. Новый метод имеет иной возвращаемый тип объекта, который сохраняется в переменной с другим идентификатором. Использование старого идентификатора также необходимо заменить на новый с учетом его особенностей, что и происходит в следующем цикле. В данном примере миграция осуществляется просто, учитывая, что у данных методов одинаковое количество аргументов и из возвращаемого типа возможно извлечь необходимую информацию, чтобы сохранить семантическую корректность.

```
ConnectivityManager connectivity =
    (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);
if (connectivity != null) {
    NetworkInfo[] info = connectivity.getAllNetworkInfo();
    for (int i = 0; i < info.length; i++)
        if (info[i].getState() == NetworkInfo.State.CONNECTED)
            return true;
}

ConnectivityManager connectivity =
    (ConnectivityManager)context.getSystemService(Context.CONNECTIVITY_SERVICE);
if (connectivity != null) {
    Network[] networks = connectivity.getAllNetworks();
    for (int i = 0; i < networks.length; i++) {
        NetworkInfo info = connectivity.getNetworkInfo(networks[i]);
        if (info.getState().equals(NetworkInfo.State.CONNECTED))
            return true;
    }
}
```

Листинг 1. Пример простой замены метода класса. Код до (сверху) и после (снизу) замены.
Listing 1. An example of a simple class method replacement. Code before (top) and after (bottom) replacement.

В дополнение, к этому подклассу относятся ситуации, в которых осуществляется трансформация блоков с циклами, в которых происходит доступ и изменение элементов массива или контейнера (в том случае, когда в API расширился интерфейс доступа к контейнеру и цикл можно сделать короче и эффективнее), например, с `while` на `for` или `foreach`, а также, в общем случае, трансформации операторов управления на иные с сохранением семантической эквивалентности.

В большинстве случаев преобразования программного кода, относящиеся к данному классу миграции API являются тривиальными, так как предполагают учет небольшого локального контекста места трансформации. Соответственно, такой тип трансформации можно осуществить автоматическим способом.

2.1.2 Замена библиотеки на схожую по функциональным возможностям

Данный подкласс миграции API относится к ситуации, когда метод класса, функция или часть исходного кода должны быть заменены другим методом, функцией или частью кода из новой библиотеки или программной платформы, обеспечивая сохранение семантической эквивалентности.

На листинге 2 представлен пример миграции программного кода, использующего библиотеку Java «Apache Network», предоставляющую функциональные возможности для работы с сетевыми протоколами, на библиотеку «Java Internal Network» являющуюся стандартной библиотекой языка Java и также предоставляющую функциональность для работы с сетевыми протоколами. Видно, что миграция сопровождается значительными изменениями в программном коде, поскольку функциональные возможности этих библиотек различаются.

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet(url);
HttpResponse response = client.execute(request);
InputStream content = response.getEntity().getContent();

URL urlObj = new URL(url);
URLConnection conn = (URLConnection)urlObj.openConnection();

conn.setRequestMethod("GET");
conn.setDoInput(true);
conn.connect();

InputStream content = conn.getInputStream();
```

Листинг 2. Пример замены метода класса. Код до (сверху) и после (снизу) замены.
Listing 2. Example of replacing a class method. Code before (top) and after (bottom) replacement.

Преобразования программного кода, относящиеся к данному подклассу миграции API, возможно осуществить автоматическим способом с помощью задания специального механизма соответствия (например, при помощи специализированного DSL) API одной библиотеки другой. Однако построение такого соответствия требует от разработчика, осуществляющего миграцию, углубленных знаний как новой, так и старой библиотеки, что существенно усложняет процесс миграции. Без задания специального механизма соответствия, учитывающего все особенности мигрируемых библиотек, автоматическая миграция невозможна.

2.1.3 Переименование метода класса или функции

Данный подкласс миграции API относится к ситуации, когда метод класса или функция должны быть переименованы при сохранении семантической эквивалентности.

Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом. При осуществлении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.1.4 Замена сигнатуры метода класса или функции

Данный подкласс миграции API относится к ситуации, когда сигнатура метода класса или функции должна быть изменена при сохранении семантической эквивалентности.

На листинге 3 представлен пример исходного кода, в котором необходимо выполнить замену сигнатуры метода путем удаления аргумента `mContext` у метода `setTextAppearance`.

```
if (b.getTag().equals(mSelectedNumber)) {  
    b.setTextAppearance(mContext, android.R.style.TextAppearance_Large);  
    b.setBackground().setColorFilter(null);  
}  
  
if (b.getTag().equals(mSelectedNumber)) {  
    b.setTextAppearance(android.R.style.TextAppearance_Large);  
    b.setBackground().setColorFilter(null);  
}
```

Листинг 3. Пример замены сигнатуры метода класса. Код до (сверху) и после (снизу) замены.

Listing 3. Example of replacing a class method signature. Code before (top) and after (bottom) replacement.

Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом в тривиальных случаях, когда количество параметров до и после преобразования совпадает или для вычисления новых параметров достаточно использовать небольшой локальный контекст вызова. При осуществлении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.1.5 Замена нестатического метода статическим и наоборот

Данный подкласс миграции API относится к ситуации, когда необходимо заменить тип вызова метода с нестатического на статический при сохранении семантической эквивалентности. Преобразования программного кода, относящиеся к данному подклассу миграции API, можно осуществить автоматическим способом. При выполнении трансформаций необходимо учитывать замену как определений метода, так и его использований.

2.2 Миграция без сохранения семантической эквивалентности (высокоуровневый класс миграции API)

К данному классу миграции API были отнесены найденные примеры миграции, которые изменили семантические свойства кода после ее проведения.

На листинге 4 представлен пример замены создания и использования объекта Java-класса `ListView` на создание и использование объектов другого Java-класса `RecyclerView`. После проведения трансформации семантические свойства кода будут отличаться, так как объекты представленных классов не обладают эквивалентным поведением.

Преобразования программного кода, относящиеся к данному классу миграции API, сложно осуществить автоматическим способом. Сложность миграции может заключаться в изменении иерархии типов классов, преобразовании методов и функций в другие методы и функции при наличии более одного подходящего кандидата на замену, а также в других аспектах.

```
ListView listView = (ListView) findViewById(R.id.list_view_announcement);
listView.setAdapter(new ListAdapter(AnnouncementActivity.this, list));
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick
    (
        AdapterView <?> parent,
        View view,
        int position,
        long id
    ) {
        startActivity(new Intent(AnnouncementActivity.this,
            AnnouncementContentActivity.class));
    }
});
```

```
RecyclerView mRecyclerView =
    (RecyclerView) findViewById(R.id.recycler_view_announcement);
mRecyclerView.setLayoutManager(
    new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false));
mRecyclerView.addItemDecoration(
    new DividerItemDecoration(this, LinearLayoutManager.VERTICAL));
mRecyclerView.setAdapter(new ListAdapter(this, list));
```

*Листинг 4. Пример замены использования объектов класса.
Listing 4. Example of replacing the use of class objects.*

2.3 Выводы

В табл. 1 представлено ранжирование найденных примеров кода по выделенным классам и подклассам миграции API.

Из таблицы видно, что большинство примеров (>90%) возможно осуществить с помощью автоматического (с использованием программных средств миграции) и полуавтоматического (с использованием программных средств миграции и небольшим участием разработчика, которое заключается в верификации результатов автоматической миграции или ручная доработка сложных случаев) способа миграции. Оставшиеся 8% найденных примеров, относящиеся ко высокоуровневому классу и ко второму подклассу низкоуровневого класса миграции API, требуют прямое участие разработчика или гибко настраиваемые инструменты, с помощью которых возможно задать семантическое сопоставление кода до или после миграции.

3. Обзор существующих инструментов и исследований

По рассматриваемой проблеме адаптации программных приложений под различные версии определенного API существует ограниченное количество программных решений и исследований, которые опубликованы и могут быть найдены в открытом доступе. Инструментов, применимых для миграции Java кода еще меньше, поэтому помимо инструментов миграции Java кода будут рассмотрены как мультязычные инструменты (не привязанные для миграции кода на определенном языке), так и для таких языков как C/C++/JavaScript и т.д. Подходы, используемые каждым из инструментов, можно обобщить как на язык Java, так и на любой другой объектно-ориентированный язык. Каждое программное решение или подход будет рассмотрено в порядке значимости (количества цитирований работ, описывающих его), а также с точки зрения возможности осуществления описанных в предыдущем разделе классов миграций. При обзоре особое внимание будет уделено удобству предлагаемых инструментом способов описания трансформаций кода для конечного пользователя.

Табл. 1. Ранжирование найденных примеров по выделенным классам и подклассам миграции API.
Table 1. Ranking of found examples by selected API migration classes and subclasses.

Класс миграции API	Доля от общего числа найденных миграций API	Способ миграции	Проблемы, возникающие в ходе миграции
Низкоуровневый: Замена метода класса/функции/части кода	30%	Автоматический	Отсутствуют
Низкоуровневый: Замена библиотеки	4%	Полуавтоматический или ручной	Отсутствие соответствия между старым и новым кодом
Низкоуровневый: Переименование метода класса или функции	15%	Автоматический	Отсутствуют
Низкоуровневый: Замена сигнатуры метода класса или функции	43%	Автоматический или полуавтоматический	Нетривиальные замены параметров метода или функции
Низкоуровневый: Замена нестатического метода статическим	4%	Автоматический	Отсутствуют
Высокоуровневый	4%	Полуавтоматический или ручной	Отображение метода многие ко многим, изменения иерархии типов и др.

Предполагается, что описание трансформаций на целевых языках или на схожих с ними DSL несет меньше нагрузки на пользователя, так как нет необходимости в глубоком изучении внутреннего устройства и механизмов работы инструмента или предполагаемого авторами инструмента подхода.

3.1 Обзор программных инструментов

3.1.1 Coccinelle и Coccinelle4J

Coccinelle [7] – это открытый инструмент для анализа и преобразования программ, написанных на языке C, который широко используется при разработке и выпуске новых версий ядра операционной системы Linux с 2006 года. Инструмент имеет открытый исходный код и доступен в различных версиях операционной системы Linux.

Целью разработки данного инструмента было повышение эффективности обновления кода ядра Linux при изменении API некоторых библиотек, используемых в программном коде ядра, путем выполнения определенных преобразований кода в полуавтоматическом режиме. Правила для преобразования программного кода записываются в Coccinelle с помощью скриптового языка SmPL (Semantic Patch Language – язык семантических преобразований), который имеет синтаксис, подобный синтаксису языка C. В языке SmPL итоговые преобразования целевого кода выражаются с использованием строк кода, представленных с префиксами «-» и «+». Первый префикс предназначен для удаления сопоставленного кода, а второй – для добавления нового кода. Coccinelle нечувствителен к комментариям и пробелам, а также учитывает поток управления и информацию о типах переменных, что существенно повышает возможности для описания сложных трансформаций на SmPL.

На листинге 5 приведен пример описания семантического преобразования для инструмента Coccinelle на SmPL.

Представленное описание семантического преобразования предполагает удаление объявления переменной только в том случае, когда переменная инициализируется константой и используется только в операторе возврата (оператор `return` языка C). Описание преобразования состоит из двух частей: блока с объявлениями метапеременных (в терминах текущего исследования метапеременной называется символьный идентификатор шаблона или правила трансформации, который хранит в себе информацию о синтаксической конструкции языка), каждая из которых задает тип узла и его символьное имя для сопоставления, располагающегося между двумя парами символов «@@», и блока с правилами преобразования кода после сопоставления.

```
@@
type T;
constant C;
identifier ret;
@@
- T ret = C;
... when != ret
when strict
return
- ret
+ C
```

Листинг 5. Пример описания семантического преобразования инструмента Coccinelle.
Listing 5. An example of a semantic patch of the Coccinelle tool.

В описанном примере преобразования объявлены три метапеременных с уникальными типами – `T`, `C` и `ret`. В блоке с правилами преобразования описано удаление объявления переменной `ret` в том случае, когда отсутствует ее использование до оператора `return`, а также замену возвращаемого выражения в операторе `return` на константу `C` из объявления переменной `ret`. Непосредственно в коде, переменная языка C может иметь любой символьный идентификатор, значение константного инициализатора, а также тип. Идентификаторы `T`, `C` и `ret` необходимы лишь для ссылки на эти сущности в правилах преобразования.

Одним из недостатков инструмента Coccinelle является то, что он может преобразовывать только код внутри тела какой-либо функции языка C, что сильно сужает его возможности, касающиеся полной миграции исходного кода между различными версиями API. Подводя итог, можно заключить, что инструмент подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграции из раздела 2, в том случае, если трансформации необходимо осуществить в пределах контекста отдельных определений функций.

Coccinelle4J – это инструмент, описанный в работе [8], являющийся надстройкой над инструментом Coccinelle. Он предназначен для трансформации кода на языке Java и обладает такими возможностями как переименование метода или изменения его сигнатуры путем добавления параметров. Coccinelle4J использовался для миграции программного кода Java между различными устаревшими API операционной системы Android.

Синтаксис языка Java схож с синтаксисом языка C, что и позволило разработать Coccinelle4J на основе Coccinelle. Таким образом, Coccinelle4J использует тот же язык семантических преобразований – SmPL. Кроме того, разработчики добавили возможность работы с блоками обработки исключений (блоки `try-catch`) и с подтипами классов.

Инструмент Coccinelle4J, как и Coccinelle, может осуществлять преобразования кода только в контексте определения отдельных методов и не поддерживает трансформацию полей

классов. Вышеперечисленное ограничивает возможность использования данного инструмента для высокоуровневой миграции исходного кода между различными версиями API, но делает пригодным использование инструмента для осуществления большинства низкоуровневых миграций из классификации раздела 2.

3.1.2 PATL

PATL (Patch-Like Transformation Language) [9] – это язык трансформаций, предназначенный для описания преобразований программного кода, написанного на языке Java. Его синтаксис схож с синтаксисом языка семантических преобразований SmPL, используемого в инструменте Coccinelle, описанном в предыдущем разделе.

На листинге 6 представлено описание правила трансформации кода на PATL.

```
// rule rButton
(jb : JButton -> Button, parent : JPanel -> Composite) {
-  jb = new JButton();
-  parent.add(jb);
+  jb = new Button(parent, SWT.PUSH);
}
```

*Листинг 6. Правила трансформации кода на PATL.
Listing 6. Rules for transforming PATL code.*

Данная трансформация подразумевает замену композиции создания объекта класса JButton и вызова метода add, помеченных знаком «-», на вызов конструктора класса Button, помеченного знаком «+».

Перед применением трансформаций выполняется преобразование кода в SSA-форму, в которой каждой переменной значение присваивается только один раз. Код, представленный в SSA-форме, обладает необходимыми свойствами для выполнения оптимизаций и упрощает анализ потока данных. Эта нормализация используется для проверки правил преобразования на согласованность друг с другом перед проведением трансформаций, а также для более точного сопоставления с правилами преобразования. PATL применяет правило сопоставления «многие ко многим»: последовательность вызовов из старого API сопоставляется с последовательностью вызовов из нового API.

Основным преимуществом PATL по сравнению с Coccinelle4J является нормализация исходного кода, что позволяет выполнять более точные сопоставления с правилами трансформации путем семантического анализа для вывода типов. Однако, с другой стороны, исходный код программы полностью преобразуется в трехадресную SSA-форму, что влечет за собой дополнительные сложности для разработчика, связанные с необходимостью обратной трансляции преобразованного представления в исходный код.

Подводя итог, можно заключить, что инструмент подходит для осуществления как низкоуровневых, так и высокоуровневых миграций из классификации раздела 2, учитывая тот факт, что трансформации происходят над преобразованным представлением, а не над исходным кодом.

3.1.3 Twinning и SWIN

В работе [10] представлен язык Twinning для описания правил трансформаций программного кода, написанного на Java. Общий вид правил трансформации на языке Twinning представлен в листинге 7. В приведенном фрагменте показано, что тип $T1_i$ должен быть сопоставлен типу $T2_i$ для всех индексов i , где каждому типу $T1_i$ сопоставляется его тип-пара $T2_i$. Каждый идентификатор x_i задает метапеременную, которая имеет тип $T1_i$ в исходном коде и отображается в экземпляра типа $T2_i$ – y_i . Шаблон выражения, которое необходимо заменить

– `JavaExp1` является выражением типа `T10`, которое может использовать метапеременные с идентификаторами `x1...xn`. `JavaExp2` задает выражение после трансформации с типом `T20`, в котором метапеременные `y1` являются результатом отображения метапеременных с идентификаторами `x1...xn`.

Пример правила трансформации на языке *Twinning*, представлен в листинге 8.

Данное правило задает правила замены всех вызовов метода `elements()` класса `Hashtable` и на вызовы метода `values().iterator()` класса `HashMap`. Например, на листинге 9 представлен фрагмент кода (слева), который преобразуется во фрагмент кода (справа).

Несмотря на то что *Twinning* учитывает правила вывода Java-типов и требует отображения типов один в один, он все еще небезопасен с точки зрения преобразования типов. Пример на листинге 10 иллюстрирует это.

```
[
T10(T1i xi, ..., T1n xn) { return JavaExp1; }
T20(T2i yi, ..., T2n yn) { return JavaExp2; }
]
```

Листинг 7. Общий вид правила трансформации кода на *Twinning*.
Listing 7. General view of the code transformation rule on *Twinning*.

```
[
Enumeration (Hashtable x)
{ return x.elements ();}
Iterator (HashMap x)
{
return x.values().iterator ();
}
]
```

Листинг 8. Правило трансформации кода на *Twinning*.
Listing 8. *Twinning* code transformation rule.

```
void f(Hashtable t ) {
Enumeration e = t.elements () :
...
}
```

```
void f( HashMap t) {
Iterator e = t.values().iterator() :
...
}
```

Листинг 9. Пример преобразования метода. Код до (слева) и после (справа) преобразования.
Listing 9. Example of method transformation. Code before (left) and after (right) conversion.

```
[
Container() { return new Container();}
Composite() { return new Composite ( new Shell(), 0);}
]
[
JList() { return new JList(); }
List() { return new List(); }
]
```

Листинг 10. Пример небезопасной трансформации типов.
Listing 10. An example of unsafe type transformation.

Описанные выше правила трансформации при применении к коду `Container x = new JList();` приведут к ошибке типов после трансформации: `Composite x = new List();`. Полученный фрагмент кода содержит ошибку, так как `JList` является подклассом класса

Container, но List таковым не является, что делает невозможным присвоение созданного объекта класса List переменной типа Composite.

Рассмотренный пример показывает, что, несмотря на то что Twinning обеспечивает корректную замену выражений и преобразование отдельных типов, применение этих замен приведет может привести к нарушению семантической корректности результирующего кода. Также инструмент не позволяет менять последовательности инструкций и блоки кода, а только лишь отдельные выражения. Этот фактор имеет существенное значение при выборе инструмента для миграции программного кода под различные версии API.

В работе [11] представлен безопасный с точки зрения преобразования типов язык **SWIN**, предназначенный для описания трансформаций программного кода, написанного на Java, который расширяет функциональные возможности инструмента Twinning. Одним из главных преимуществ SWIN по сравнению с Twinning является безопасность преобразований типов объектов.

Программа на языке SWIN состоит из трех частей: правила преобразования, шаблона исходного кода и шаблона кода после преобразований. Пример использования SWIN описан ниже. Исходный код на Java:

На листинге 11 представлена программа преобразования SWIN, где 1 и 2 – это правила преобразования.

На листинге 12 представлен исходный код на языке Java до (слева) и после (справа) преобразования с применением правила из листинга 7.

```
P = [1, 2]
where
1 = () [ new A() : A -> new B() : B ]
2 = (x : A -> B, u : A -> B)
[ x.h(u) : A -> x.k(u, new B()) : B ]
```

Листинг 11. Пример программы преобразования SWIN.
Listing 11. Example of SWIN conversion program.

(new A()).h(new A())	(new B()).k(new B(), new B())
----------------------	-------------------------------

Листинг 12. Пример трансформации выражения на языке Java. Код до (слева) и после (справа) преобразования.

Listing 12. Example of transforming an expression in Java. Code before (left) and after (right) conversion.

Данное правило трансформации описывает замену объектов класса A на объекты класса B, а также замену метода h класса A на метод k класса B в программном коде.

Несмотря на все преимущества языка SWIN, как и Twinning, он позволяет трансформировать лишь отдельные выражения. Это существенно ограничивает его применимость для миграций больших кодовых баз под различные версии API. Также, в отличие от Twinning, описания трансформаций для SWIN обладают гораздо более нетривиальным синтаксисом, что создает дополнительную нагрузку на пользователя.

Подводя итог, можно заключить, что SWIN, как и Twinning, подходят для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью них возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.4 Nobrainer

Инструмент Nobrainer [12] разработан на основе программного обеспечения компилятора Clang [13] из компиляторной инфраструктуры LLVM [14] в Институте системного программирования им. В.П. Иванникова РАН. Он предназначен для модификации кода, 122

написанного на языках C/C++, с использованием специальных правил преобразования кода. Правила для Nobrainer пишутся на C/C++ без использования специализированных языков (DSL, Domain-Specific Language – предметно-ориентированный язык). На листинге 13 представлен пример правил преобразования для инструмента Nobrainer.

```
int NOB_BEFORE_EXPR(ChangeOrder)(Agent a, char *x, char *y) {  
    return a.compose(x, y);  
}  
int NOB_AFTER_EXPR(ChangeOrder)(Agent a, char *x, char *y) {  
    return a.compose(y, x);  
}
```

Листинг 13. Пример правил преобразования Nobrainer.

Listing 13. An example of Nobrainer transformation rules.

В данном примере показано правило преобразования, которое изменяет порядок аргументов внутри вызова метода `compose` класса `Agent`.

Шаблоны трансформации, представляющие фрагмент целевого кода до и после преобразования, пишутся как отдельные функции. Для сопоставления кода при проведении трансформаций Nobrainer использует AST-сопоставители (Abstract Syntax Tree – абстрактное синтаксическое дерево) Clang [15] для поиска подходящих шаблонов по дереву абстрактного синтаксиса.

Nobrainer применяет все замены или создает файл в формате YAML [16] с описанием этих замен. Процесс работы Nobrainer состоит из трех основных этапов:

1. Анализ всех единиц трансляции кода, переданных инструменту командами компиляции, и фильтрация шаблонов сопоставления на действительные и недействительные.
2. Проверка каждого правила на предмет согласованности и его обработка для создания внутреннего представления шаблона преобразования.
3. Сопоставление каждого правила с исходным кодом программы.
4. Генерация замен по шаблону.

Nobrainer также поддерживает написание шаблонизированных правил (см. листинг 14).

```
template <class T> T *before() {  
    return (T *)malloc(sizeof(T));  
}  
template <class T> T *after() {  
    return new T;  
}
```

Листинг 14. Пример шаблонизированных правил Nobrainer.

Listing 14. An example of Nobrainer template rules.

Данный пример демонстрирует ситуацию, когда разработчику необходима частичная миграция, связанная с динамическим выделением памяти, со стиля языка C на C++.

Несомненным плюсом Nobrainer является формат описания преобразований непосредственно на целевых языках (C/C++), что упрощает использование данного инструмента целевым пользователем. Однако, несмотря на все функциональные возможности инструмента Nobrainer, его формат описания правил преобразования в виде функций и их возвращаемых значений не охватывает значительную часть конструкций целевых языков C/C++, а также не позволяет менять блоки кода единым правилом трансформации (для этого придется писать множество косвенно связанных отдельных правил). Например, с помощью данного инструмента невозможно заменять конструкции

циклов `for(...)` на `while(...)`, что является ограничением при выборе инструмента для миграции кода между различными версиями API.

Подводя итог, можно заключить, что инструмент Nobrainer подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.5 MARTINI

MARTINI [17] – это инструмент для анализа и преобразования программ на языках C/C++, который использует подход инструмента Nobrainer. В MARTINI задаются пары шаблонов поиска (`match`) и шаблонов трансформации (`replace`) соответственно. В отличие от Nobrainer, который позиционируется как безопасный инструмент с точки зрения преобразования типов, разработчики MARTINI используют менее строгий синтаксис описания шаблонов. Например, символьные имена внутри шаблона не воспринимаются как реально существующие в исходном коде. Это означает, что в Nobrainer (см. раздел 3.1.4) невозможно заменить тип произвольного выражения с произвольными именами переменных, тогда как подход MARTINI это позволяет.

На листинге 15 представлен пример кода шаблонов поиска и трансформации, которые заменяют инициализацию указателя произвольного типа с `0` на `nullptr`.

```
template <typename T>
[[clang::matcher("nullptr-decl")]]
auto null_match()
[[clang::matcher_block]]
T *var = 0;
}

template <typename T>
[[clang::matcher("nullptr-decl")]]
auto null_replace()
[[clang::matcher_block]]
T *var = nullptr;
}
```

Листинг 15. Пример шаблона трансформации MARTINI.
Listing 15. Example of MARTINI transformation template.

На данный момент инструмент поддерживает только ограниченное подмножество стандартов C/C++ и находится в активной стадии разработки, что является существенным ограничением при выборе инструмента для миграции кода между различными версиями API. Подводя итог, можно заключить, что MARTINI подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.6 Refaster

Refaster [18] – это открытый инструмент, разработанный компанией Google (впоследствии переименованный в Error-Prone), предназначенный для рефакторинга Java кода с помощью шаблонов кода «до» и «после» его сопоставления. Шаблоны записываются на языке Java без специализированного DSL.

В качестве примера рассмотрим перехода на новый API библиотеки кодирования Base64. На листинге 16 представлен код до и после преобразования.

Правило преобразования кода инструмента Refaster, позволяющее осуществить данную миграцию, представлено на листинге 17.

Правила (шаблоны) преобразования, применяемые инструментом Refaster, включают в себя класс, содержащий один или несколько аннотированных методов с пометкой `@BeforeTemplate` и один метод, аннотированный как `@AfterTemplate`. Каждый из методов шаблона может содержать аргументы, именуемые переменными выражений, которые соответствуют выражениям определенного типа в пользовательском коде; например, это могут быть байты, связанные с кодированием *Base64*. Тело метода шаблона представляет собой выражение, возвращаемое оператором `return`, аналогично подходу, используемому в инструменте Nobrainer (см. раздел 3.1.4). Шаблоны преобразования Refaster описывают замену любого выражения в исходном коде, которое соответствует одному из выражений, возвращаемых методами `@BeforeTemplate`, на выражение, возвращаемое единственным методом `@AfterTemplate`.

Однако одним из основных недостатков Refaster является отсутствие семантического анализа (хотя бы для корректного вывода типов) кода перед сопоставлением. Приведенный на листинге 18 фрагмент кода не будет преобразован в соответствии с установленным правилом.

```
System.out.println(Base64.encodeWebSafe(Files.toByteArray(file), false));

System.out.println(BaseEncoding.base64Url().omitPadding().encode(Files.toByteArray(file)));
```

Листинг 16. Пример миграции библиотеки кода, использующего библиотеку *Base64*.
Код до (сверху) и после (снизу) миграции.

Listing 16. An example of migrating a code library that uses the *Base64* library.
Code before (top) and after (bottom) migration.

```
class BaseEncodingMigration {
    @BeforeTemplate public String before(byte[] bytes) {
        return Base64.encodeWebSafe(bytes, false /* no padding */);
    }
    @AfterTemplate public String after(byte[] bytes) {
        return BaseEncoding.base64Url().omitPadding().encode(bytes);
    }
}
```

Листинг 17. Пример правила трансформации Refaster.
Listing 17. An example of a Refaster transformation rule.

```
doPadding = false;
System.out.println(Base64.encodeWebSafe(Files.toByteArray(file), doPadding));
```

Листинг 18. Пример правила трансформации Refaster.
Listing 18. An example of a Refaster transformation rule.

В данном случае инструмент Refaster не способен определить, что переменная `doPadding` имеет булевый тип. Так как он учитывает только типы стандартной библиотеки языка Java (примитивные и классы), а также классы из Java библиотек от компании Google.

Более того, Refaster иногда стремится переписать реализацию нового API. Например, если метод `BaseEncoding.base64Url(...)` в приведенных ранее правилах Refaster использует в своей реализации метод `Base64.encodeWebSafe(...)`, это может привести к возникновению бесконечного цикла, когда Refaster попытается переопределить `BaseEncoding.base64Url(...)` в терминах самого себя. Таким образом, инструменту Refaster необходима помощь разработчика для разрешения определенных ситуаций, связанных с преобразованием кода, которые он не в состоянии решить самостоятельно.

Указанные выше ограничения инструмента Refaster препятствуют выполнению сложных и масштабных преобразований кода, требующих семантический анализ сопоставленных конструкций. В дополнение к этому, Refaster не способен трансформировать цепочки операторов, а лишь отдельные выражения. В связи с этим, Refaster подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов, но не их определений.

3.1.7 Proteus

Proteus [19] – это проприетарный инструмент, предназначенный для анализа и преобразования программ на языке C/C++, учитывающий пробельные символы и комментарии. Инструмент использует промежуточное представление программы LL-AST (Literal-Layout AST), которое учитывает количество пробельных символов, табуляции, пользовательские комментарии, а также отдельные литералы, такие как «(», «)» в качестве самостоятельных элементов абстрактного синтаксического дерева.

Шаблоны поиска в Proteus описываются с помощью разработанного DSL YATL (Yet Another Transformation Language), который синтаксически схож со Stratego [20].

Разработанный язык позволяет захватывать узлы LL-AST и сопоставлять их внутренним переменным языка YATL для дальнейшего семантического анализа. Язык YATL позволяет использовать конструкции целевого языка в задании типов узлов AST, которые необходимо сопоставить и трансформировать, а также ссылаться в шаблонах трансформации на нетипизированные переменные языка YATL, хранящие в себе значения AST узлов.

Пример шаблона поиска и трансформации вызова функции `boo` на вызов функции `foo` представлен на листинге 19.

```
foreach-match(FunctionCall:{'boo',=*p}) {  
  on *p {  
    $_ = new(Id:{'foo'});  
  }  
}
```

Листинг 19. Пример правила трансформации Proteus.

Listing 19. An example of a Proteus transformation rule.

Фактически, трансформации кода, выполняемые Proteus представляют собой интеллектуальные макроподстановки, а сам инструмент представляет собой препроцессор, учитывающий комментарии, пробельные символы и отступы, но, несмотря на это, инструмент не поддерживает трансформации, требующие семантического анализа, например, с помощью него невозможно осуществить предварительную проверку типов перед сопоставлением и заменой и прочее. Также Proteus обладает нетривиальным синтаксисом языка описания правил трансформации, что может потребовать от пользователя, осуществляющего трансформацию, дополнительных навыков и времени на его изучение.

Инструмент подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.8 SPOON

В работе [21] представлена открытая библиотека SPOON, разработанная на языке программирования Java. Эта открытая библиотека предназначена для анализа и преобразования кода программ, написанных на языке Java, и может быть использована для

различных преобразований, включая изменения классов, методов и типов переменных. В процессе анализа исходного кода библиотека SPOON создает абстрактное синтаксическое дерево (AST), использующее информацию о метамодели объектов SPOON, являющейся расширением стандартной метамодели языка Java, которая задает иерархию сущностей (например, метод, класс, переменная и т. д.) Java с их атрибутами и является фактически структурой Java AST. Метамодель SPOON включает специально разработанные классы, имеющие префикс Ct. Например, CtElement является базовым (корневым) классом метамодели SPOON, CtClass представляет абстракцию класса Java, а CtMethod служит абстракцией для объявления метода Java.

Преобразование программы в контексте библиотеки SPOON предусматривает комбинацию её анализа и последующей трансформации. Эта концептуальная пара реализована в классе-обработчике SPOON, который фокусируется на анализе элементов программы в метамодели. В качестве иллюстрации, на листинге 20 приведен код, демонстрирующий процессор, который анализирует программу с целью поиска и замены метода getText класса A на новый метод testReplace.

```
public class MethodProcessor extends AbstractProcessor<CtElement> {
    public void process(CtElement el) {
        if (el instanceof CtMethod<?>) {
            // Замена объявления метода
            CtMethod<?> method = (CtMethod<?>)el;
            if (... && method.getSimpleName().equals("getText")) {
                method.setSimpleName("testReplace");
            }
            // Замена вызовов метода
            if (el instanceof CtInvocation<?>) {
                ...
            }
        }
    }
}
```

Листинг 20. Пример класса-обработчика SPOON.

Listing 20. An example of SPOON processor.

В данном примере захватываются все узлы AST, так как каждый узел является производным от базового класса метамодели SPOON – CtElement. Метод process класса MethodProcessor принимает запрошенный элемент в качестве входных данных и выполняет анализ, в результате чего происходит замена имени метода getText класса A при объявлении и во всех вызовах. Для использования обработчика требуется его добавление в библиотеку SPOON. В библиотеке допускается использование нескольких обработчиков одновременно, при этом порядок их применения определяется пользователем.

Тестирование показало, что библиотека SPOON хорошо распознает цепочки вызовов функций. Например, она может распознать и изменить метод getText Java-класса A с помощью описанного выше обработчика в ситуации на листинге 21.

```
Class c = new C();
c.getA().getText()
```

Листинг 21. Пример трансформируемого Java кода.

Listing 21. An example of Java programming code

SPOON способен распознать, что возвращаемое значение метода – c.getA() является экземпляром класса A.

Однако данная библиотека не предоставляет функциональности для работы со статическими методами классов, а также не учитывает поток управления программы при трансформациях

кода (анализ потока управления при использовании данного инструмента можно провести вручную, обходя дерево AST самостоятельно).

Среди других возможностей библиотека SPOON предоставляет возможности для модификации не только исходного пользовательского кода, но и пользовательских комментариев. Ограничениями SPOON являются классы Java, которые определены во внешних пакетах Java. В контексте простых примеров трансформации кода, таких как преобразование классов, методов и переменных, библиотека SPOON демонстрирует хорошие результаты.

Библиотека SPOON используется во многих инструментах восстановления и преобразования Java-кода, включая:

1. Genesis [22] – инструмент для автоматического исправления кода Java (добавление или замена предикатов в блок с условием оператора `if`, замена операторов возврата `return`, а также других типов ошибок);
2. Elixir [23] – инструмент для поиска дефектов в программном обеспечении, написанном на языке программирования Java;
3. Cap-Gen [24] – инструмент для автоматической генерации некоторого ограниченного класса контекстно-зависимых преобразований;
4. MuCrash [25] – инструмент для выявления условий сбоя выполнения программы и генерации тестов для воспроизведения этих сбоев;
5. Nopol [26] – инструмент для автоматического исправления блоков `if-then-else`, который учитывает поток управления программой, написанной на языке программирования Java;
6. Astor [27] – библиотека, предназначенная для поиска дефектов в программах, написанных на языке программирования Java.

Представленные в списке выше инструменты и библиотеки имеют ограниченную сферу применения, косвенно связанную с миграцией программного кода, соответственно, инструментами трансформации программного кода они не являются.

SPOON подходит для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2.

3.1.9 Comby

Comby [28] – это инструмент для синтаксической трансформации кода, который использует концепцию парсеров-комбинаторов (Parser Combinators) для многократной обработки синтаксических структур языков программирования. В работе авторы подробно описывают архитектуру инструмента, его возможности и применяемые методы.

Comby предоставляет контекст для выполнения трансформаций на основе синтаксических шаблонов и обладает поддержкой множества языков программирования. Инструмент поддерживает описание правил трансформации на языке, схожим с целевым.

Функциональные возможности инструмента Comby включают в себя системы поиска и замены, что позволяет разработчикам эффективно выполнять рефакторинг, обновления API и другие виды преобразования программного кода.

В листинге 22 снизу показан пример замены цикл `for` на языке Python, который вычисляет сумму, на функцию `np.sum()` из API библиотеки NumPy. Преобразование описывается правилом, показанным в листинге 8, в котором левая часть задает шаблон «до», а правую шаблон «после». Обе стороны правила содержат операторы Python с переменными шаблона (например, `:[v0]`), которые связываются с узлами AST фактического исходного кода (например, `:[v0]` связывается с результатом вычисления). Эти правила могут быть использованы для преобразования любого целевого кода, который имеет аналогичную

структуру AST с кодом, представленным в листинге 8, независимо от различий в именах переменных.

Хотя Comby поддерживает множество языков и обладает синтаксисом шаблонов трансформации схожим с целевыми языками (что повышает удобство его использования), его эффективность снижается при работе с проектами, обладающими крупными кодовыми базами. Так как Comby не проверяет синтаксическую и семантическую корректность кода после подстановки, то в результате преобразований может получиться некорректный код, например, в шаблон поиска цикла из листинга 22 может попасть цикл, итерируемый по строке. Это приведет к тому, что в `numpy.sum` окажется строка из переменной шаблона – `: [[v2]]`, и возникнет ошибка несоответствия интерфейсов типов времени выполнения интерпретатора `python`. Соответственно, чем больше кодовая база, тем больше будет таких ошибок. В таком случае потребуется ручная верификация результатов преобразований, что не сильно проще осуществления преобразований вручную. Также не получится задать дополнительные семантические проверки (например, типа) переменных шаблона, содержащих сопоставляемые синтаксические конструкции, так как Comby рассчитан лишь на проведение простых трансформаций в виде поиска и замены. Переменные шаблона представляют собой лишь найденные синтаксические конструкции в исходном коде, а не являются типизированными AST узлами.

<pre>: [[v0]] = 0 for : [[v1]] in : [[v2]]: : [[v0]] = : [[v0]] + : [[v1]]</pre>	<pre>: [[v0]] = numpy.sum(: [[v2]])</pre>
<pre>- result = 0 - for elem in elements: - result = elem + result + result = numpy.sum(elements)</pre>	

Листинг 22. Пример использования Comby.

Правило перезаписи «до» (слева-сверху) и «после» (справа-сверху), Итоговое преобразование (снизу).

Listing 22. An example of Comby tool usage.

The rewriting rule "before" (left-top) and "after" (right-top), the final transformation (bottom).

Из-за перечисленных недостатков Comby подходит для осуществления тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API, так как с помощью него возможно осуществить лишь трансформацию вызовов методов и функций, но не их определений.

3.1.10 OpenRewrite

OpenRewrite [29] – это инструмент с открытым исходным кодом, предназначенный для рефакторинга и миграции программного кода на языке Java. Правила миграции можно записывать как в декларативном, так и в императивном стиле.

Декларативный стиль предполагает использование существующих правил миграции с параметризацией, которые описываются в файле конфигурации на языке YAML. В листинге 23 представлен пример использования такого файла конфигурации, предполагающего миграцию с библиотеки Java ApacheClient [30] версии 4 на версию 5. Такой тип миграции предполагает существенное изменение описаний классов и их методов, соответственно используются разные правила миграции, каждая из которых осуществляет определенный ее вид.

OpenRewrite также позволяет описывать правила миграции выражений или отдельных операторов в императивном стиле с помощью шаблонов `@BeforeTemplate` и `@AfterTemplate` библиотеки инструмента Refaster (см. раздел 3.1.6).

Инструмент поддерживает также более интеллектуальный тип миграций с использованием обходчиков LST (Lossless Semantic Tree – семантического дерева без потерь) [31], который представляет собой вариант реализации AST с расширенной информацией о типе каждого узла дерева и сохранением информации о форматировании исходного кода (наличие отступов и комментариев). На листинге 24 представлен пример описания правила миграции с помощью Java шаблонов.

```
---
type: specs.openrewrite.org/v1beta/recipe
name: org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5
displayName: Migrate to ApacheHttpClient 5.x
description: >
  Migrate applications to the latest Apache HttpClient 5.x release.
recipeList:
  - org.openrewrite.java.apache.httpclient4.UpgradeApacheHttpClient_4_5
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_ClassMapping
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_DeprecatedMethods
  - org.openrewrite.java.apache.httpclient5.UpgradeApacheHttpClient_5_TimeUnit
  - org.openrewrite.java.apache.httpclient5.StatusLine
---
```

Листинг 23. Пример декларативного шаблона миграции OpenRewrite.
Listing 23. An example of a declarative OpenRewrite migration pattern.

```
@Value
@EqualsAndHashCode(callSuper = false)
public class SayHelloRecipe extends Recipe {
    ...
    public class SayHelloVisitor extends JavaIsoVisitor<ExecutionContext> {
        private final JavaTemplate helloTemplate =
            JavaTemplate.builder("public String hello() { return \"Hello from #{}!\"; }")
                .build();

        @Override
        public J.ClassDeclaration visitClassDeclaration(...) {
            ...
            // Проверка того, что в классе нет метода "hello"
            boolean helloMethodExists = classDecl.getBody().getStatements().stream()
                .filter(statement -> statement instanceof J.MethodDeclaration)
                .map(J.MethodDeclaration.class::cast)
                .anyMatch(methodDeclaration ->
                    methodDeclaration.getName().getSimpleName().equals("hello"));

            ...
            // Добавление метода в описание класса
            classDecl = classDecl.withBody(helloTemplate.apply(...));
            return classDecl;
        }
    }
}
```

Листинг 24. Пример императивного шаблона миграции OpenRewrite.
Listing 24. An example of an imperative OpenRewrite migration pattern.

В примере выше описывается правило, добавляющее в каждый найденный класс метод hello, если метод с таким наименованием в этом классе отсутствует. Правило миграции записывается в виде Java класса, производного от Recipe, который предоставляет интерфейс, позволяющий задать имя шаблону миграции, текстовое описание, способ сериализации, а также набор обходчиков LST. В данном примере зарегистрирован всего один обходчик – SayHelloVisitor, в котором представлен обработчик объявлений класса,

проверяющий, что в классе присутствует метод с именем `hello` и добавляющий его в противном случае.

Созданный таким образом шаблон затем можно использовать в файлах конфигурации YAML вместе с другими шаблонами для проведения более сложных трансформаций.

OpenRewrite предоставляет заранее разработанные шаблоны миграции Java кода в виде библиотеки. Однако результат его работы требует дополнительного анализа, так как OpenRewrite не гарантирует семантическую корректность преобразованного кода. Также эффективность работы инструмента для определенного кода или проекта зависит от качества и полноты доступных в библиотеки шаблонов миграции. Для нетривиальных миграций имеющихся шаблонов не будет достаточно, что потребует от разработчика изучения библиотеки OpenRewrite для написания новых и интеграции их с существующими. Учитывая необходимость верификации результатов трансформации, изучения библиотеки шаблонов и написания новых, затраты времени на процесс миграции могут быть существенны для больших кодовых баз, что ставит под сомнение возможность использования OpenRewrite вместо осуществления миграций вручную.

Несмотря на это библиотека OpenRewrite позволяет описывать сложные трансформации с помощью семантического анализа, используя информацию о типах и взаимосвязь узлов из LST, что автоматически делает его использование пригодным для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2, а также с помощью него возможно осуществлять высокоуровневые миграции.

3.1.11 GoPatch

Открытый инструмент GoPatch [32] представляет собой утилиту командной строки для простого рефакторинга кода на языке Go. Правила трансформации описываются с помощью шаблонов (см. листинг 25), с использованием собственного DSL, похожего на формат Unified Diff [33].

```
# Replace time.Now().Sub(x) with time.Since(x)
@@
var x identifier
@@

-time.Now().Sub(x)
+time.Since(x)
# Not a description comment
```

Листинг 25. Пример правила трансформации GoPatch.
Listing 25. Example of a GoPatch transformation rule.

Правило на листинге 25 позволяет найти и заменить вызовы `time.Now().Sub(x)` вызовами `time.Slice(x)`, где `x` является метапеременной шаблона трансформации с типом идентификатора. GoPatch предоставляет два доступных типа метапеременных для использования в шаблонах – идентификатор и выражение. Поиск и сопоставление осуществляется по AST исходного кода на языке Go. Ищутся строки, помеченные «-», при этом осуществляется попытка проверки соответствия типов, вложенных в строку метапеременных. При успешном сопоставлении генерируются строки, помеченные «+», в которых раскрываются извлеченные значения метапеременных в качестве замены сопоставленным строкам шаблона. После обработки всего AST генерируется описание трансформации кода в формате Unified Diff.

Использование такого формата DSL является удачным решением, так как шаблоны трансформации описываются достаточно просто с использованием целевого языка. Однако инструмент GoPatch не позволяет производить трансформации, требующие семантического

анализа сопоставленных узлов перед их заменой. Например, трансформации, требующие проверку переменных на соответствие определенному типу до применения преобразований к коду, использующему их. Соответственно он подходит для осуществления большинства миграций только подклассов (кроме второго) низкоуровневого класса миграций API из раздела 2.

3.1.12 CodeTurn

Astadia CodeTurn [34] является инструментом с закрытым исходным кодом, который осуществляет миграции программного кода на языке Cobol на объектно-ориентированные (ООП) языки высокого уровня – Java и C#. Инструмент был разработан для осуществления миграций старых кодовых баз на языке Cobol на более распространенные языки с целью упрощения их поддержки и развития. CodeTurn позволяет мигрировать код, используя как процедурные, так и объектно-ориентированный стиль целевого языка программирования, например, Java. Примеры миграции кода на языке Cobol с использованием CodeTurn представлены на листинге 26.

COBOL	<pre>SELECT DATFILE ASSIGN TO "test.dat" * ... OPEN OUTPUT DATFILE ACCEPT W-TIME FROM TIME MOVE SS OF W-TIME TO BREAK-VAL WRITE DAT-REC</pre>
Процедурная Java	<pre>cobol.open(datfile, "test.dat", FileOpenMode.OUTPUT); cobol.acceptTime(wTime); cobol.move(wTime.ss, datfile.datrec.breakVal); cobol.write(datfile);</pre>
ООП Java	<pre>datfile.open("test.dat", FileOpenMode.OUTPUT); wTime.setValue(cobol.getTimeAsString()); datfile.datrec.breakVal.setValue(wTime.ss); datfile.write();</pre>

Листинг 26. Миграция кода на COBOL с использованием CodeTurn. Исходный код на COBOL (сверху), Java в процедурном стиле после миграции (посередине), Java в ООП стиле после миграции (снизу).
Listing 26. Migrating code to COBOL using CodeTurn. COBOL source code (top), Procedural style Java after migration (middle), OOP style Java after migration (bottom).

Несмотря на широкую функциональность инструмента, CodeTurn не позволяет мигрировать произвольный код на Cobol, так как поддерживаются не все виды операторов данного языка, что потребует их ручной миграции после проведения автоматической трансформации. CodeTurn подходит для осуществления большинства миграций низкоуровневого класса миграций API из раздела 2.

3.1.13 TXL

TXL [35] – это специализированный язык программирования, предназначенный для быстрого создания прототипов описаний грамматик языков программирования, а также для выполнения различных преобразований с данными языками. С помощью него можно описывать правила преобразования в том числе и объектно-ориентированных языков.

Программы на языке TXL состоят из трех частей:

1. Не зависящей от контекста базовой грамматики для языка, с которым необходимо выполнять преобразования;
2. Набора не зависящих от контекста грамматических преобразований (расширений или изменений) базовой грамматики языка;
3. Корневого набора преобразований в базовый язык.

Пример правила преобразования операции синтаксически длинного присваивания в операцию синтаксически короткого присваивания, описанного на языке TXL представлен на листинге 27.

Применимость TXL к различным классам миграции из раздела 2 существенно зависит от того или иного инструмента, которые реализует, описанные в работе авторов TXL механизмы. Но, подводя итог, можно заключить, что с помощью подхода авторов TXL возможно осуществлять лишь тривиальные миграции только из низкоуровневого класса, так как язык ориентирован на грамматические преобразования без учета семантического смысла конструкций целевого языка, а также их типов. Подводя итоги, TXL подходит для описания лишь тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API.

```
rule simplifyAssignments
  replace [statement]
    V [reference] := V + E [term]
  by
    V += E
end rule
```

Листинг 27. Пример правила трансформации присваивания на TXL.

Listing 27. Example of assignment transformation rule in TXL.

3.1.14 JScodeshift

JScodeshift [36] – это открытый инструмент, разработанный компанией Facebook, предназначенный для анализа программного кода, написанного на языке JavaScript. Инструмент работает с AST представлением кода и использует JavaScript библиотеку Recast [37] для анализа и изменения AST дерева программы.

Еще одной особенностью JScodeshift является использование модуля AST-Types, который предоставляет абстрактную иерархию типов синтаксического дерева (AST), впервые примененную в Mozilla Parser API [38].

Правила миграции программного кода для инструмента описываются на языке программирования JavaScript и оформляются в виде функций. На листинге 28 приведен пример миграции программного кода JavaScript библиотеки AvaJs4 [39] средствами JScodeshift.

На листинге 28 свойство класса `ok` заменяется свойством `truthy`, а `notOk` свойством `falsy`.

Другие программные платформы и библиотеки, такие как React5 [40] и Lodash6 [41], используют JScodeshift для миграции между версиями.

После выполнения преобразований инструмент JScodeshift генерирует AST дерево в формате ESTree [42], которое может быть транслировано обратно в высокоуровневый программный код с использованием API компилятора языка TypeScript [43]. Ограничением инструмента является отсутствие возможности добавления семантических проверок свойств узлов AST (например, проверки корректности преобразования объекта к определенному типу) до и после сопоставления, соответственно он подходит для осуществления лишь тривиальных миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций API.

```
function renameAssertion(name, newName, j, ast) {
  ast
    .find(j.CallExpression, {callee: { object: {name: 't'}, property: {name}}})
    .forEach(p => p.get('callee')
      .get('property')
      .replace(j.identifier(newName)));
}
```

```
}  
export default function (file, api) {  
  const j = api.jscodeshift;  
  const ast = j(file.source);  
  renameAssertion('ok', 'truthy', j, ast);  
  renameAssertion('notOk', 'falsy', j, ast);  
  return ast.toSource();  
}
```

Листинг 28. Пример правила миграции библиотечного кода средствами JSCodeshift.

Listing 28. An example of a library code migration rule using JSCodeshift.

3.1.15 Rascal

Rascal [44] – это интерпретируемый DSL (Domain-Specific Language – предметно-ориентированный язык), предназначенный для описания трансформации кода программ, написанных на Java, который имеет свой собственный набор примитивных типов данных, таких как `boolean`, `int` и других. Rascal имеет множество возможностей: сопоставление с образцом исходного кода по его AST, обработка исключений, функции, правила перезаписи и другие.

Сопоставление с образцом – это механизм поиска фрагментов кода по шаблону в Rascal. Например, следующее выражение соответствия может использоваться (см. листинг 29), чтобы находить оператор языка Java `while`.

```
whileStat (Exp cond, list [ Stat ] body) := stat
```

Листинг 29. Выражение соответствия для оператора `while`.

Listing 29. Match expression for the `while` statement.

Переменные шаблона, такие как `cond` и `body`, могут быть объявлены как в этом примере, или они могут быть объявлены в контексте, в котором будет сопоставляться шаблон. Также в Rascal определены правила перезаписи, которые являются единственным неявным механизмом управления в языке и используются для поддержки инвариантов во время вычислений. На листинге 30 представлено правило, которое меняет структуру операторов `if` языка Java.

```
rule  
  ifStat (neg(Exp cond), list [ Stat ] thenPart,  
    list [ Stat ] elsePart)  
  -> ifStat (cond, elsePart, thenPart);
```

Листинг 30. Пример правила трансформации операторов `if` на Rascal.

Listing 30. Example of a rule for transforming `if` operators in Rascal.

Если блок с условием оператора `if` содержит выражение инверсии, то инверсия из данного выражения удаляется и ветви меняются местами. Это правило будет срабатывать каждый раз, когда обнаружится оператор `if`, соответствующий левой части правила.

Но, несмотря на все преимущества языка Rascal, его синтаксис достаточно далек от синтаксиса Java. Соответственно, пользователю, осуществляющему миграцию, потребуется время для его изучения. Другим недостатком Rascal являются отсутствие возможностей проведения семантического анализа (например, проверка эквивалентности типов) перед сопоставлением, что может потребоваться при проведении высокоуровневых миграций для объектно-ориентированных языков программирования, так как Rascal ориентирован на синтаксическое сопоставление по AST узлов и не предоставляет возможности для анализа

потока данных. Соответственно, Rascal подходит для проведения большинства низкоуровневых трансформаций, описанных в разделе 2.

3.1.16 ClangMR

ClangMR [45] – это инструмент, разработанный в компании Google для упрощения миграций внутренней кодовой базы. Реализованный в инструменте подход основан на использовании в шаблонах трансформации AST сопоставителей компилятора Clang, как и в случае рассмотренного ранее инструмента Nobrainer. Основное отличие состоит в том, что написание шаблонов трансформации для ClangMR требует знаний внутреннего API Clang и структуры абстрактного синтаксического дерева. На листинге 31 приведен пример сопоставления и замены вызовов метода Bar класса Foo методом Baz из того класса.

Для осуществления миграции кода путем его трансформации, пользователь определяет один или несколько сопоставителей Clang AST (в примере на листинге 31 описан сопоставитель, предназначенный для поиска всех вызовов с именем Bar из пространства имен Foo, обладающих одним аргументом). Также пользователю необходимо определить обработчики результатов сопоставления по AST (в примере описана функция, которая проверяет, что сопоставленный вызов является вызовом метода в терминах языка C++, а затем производит замену имени вызова на Baz, путем тривиальной замены лексической единицы трансляции (replaceToken)).

```
StatementMatcher matcher =
    callExpr(allOf(
        argumentCountIs(1),
        callee(functionDecl(hasName(
            "::Foo::Bar")))))
    .bind("call");

void Refactor(const MatchFinder::MatchResult& res) {
    Clang::CXXMemberCallExpr* call =
        res.Nodes.getStmtAs<clang::CXXMemberCallExpr>(
            "call");
    const clang::MemberExpr *member_expr =
        llvm::dyn_cast<clang::MemberExpr>(
            call->getCallee());
    EditState state(res, call);
    state.ReplaceToken(member_expr->getMemberLoc(), "Baz");
    Report(&state);
}
```

Листинг 31. Сопоставитель для поиска вызовов метода Bar (сверху).

Пример реализации правила трансформации вызовов метода Bar с использованием ClangMR (снизу).

Listing 31. Matcher for finding Bar method calls (top).

An example of a handler for transforming Bar method calls using ClangMR (bottom).

Ограничениями такого подхода являются: привязанность к языку (C/C++), необходимость наличия углубленных знаний AST Clang и его API (в отличие от Nobrainer, в котором трансформации осуществляются на целевом языке), нетривиальность проведения сложных трансформаций, что влечет за собой увеличение времени процесса миграции больших кодовых баз. Несмотря на то, что библиотека Clang предоставляет широкие возможности для семантического анализа по AST (включая проверку типов и моделирование потока управления), который является необходимым при описании высокоуровневых миграций, все же у Clang есть ограничения, связанные с невозможностью анализа зависимостей между разными единицами трансляции.

Подводя итог, можно заключить, что инструмент подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу, описанному в разделе 2, а также с помощью него возможно осуществлять высокоуровневые миграции, используя функциональные возможности библиотеки Clang.

3.1.17 Другие инструменты

В обзоре не был представлен фреймворк Cubix [46], предназначенный для описания мультязычных трансформаций. Также отсутствует упоминание о недавно представленном инструменте PoliglotPirahna [47], основанном на фреймворке языковых подзапросов tree-sitter [48]. Этот инструмент развивает идеи своего предшественника – Piranha [49], который опирается на технологию Egor-Prone (см. раздел 3.1.6).

Кроме того, DMS [50] – инструмент с закрытым исходным кодом, который предлагает фронтенд для множества языков и позволяет описывать правила трансформации в декларативно-процедурном стиле с использованием собственного языка описания (DSL), обладающего сложным синтаксисом. Ограничением инструмента является необходимость уточнения и реализации семантики целевого языка для осуществления сложных трансформаций.

Отдельного внимания заслуживают инструменты, представленные в виде расширений программных средств разработки (IDE), например, такие как SemDiff [51] и Catchup! [52], которые помогают в трансформации и анализе программного кода. SemDiff является плагином для Eclipse IDE [53] и уведомляет разработчиков об изменениях API Java библиотек, используемых в проекте. В свою очередь, Catchup! также выступает в качестве плагина для Eclipse IDE и предлагает функциональные возможности для рефакторинга исходного кода Java, включая такие операции как переименование типов, перемещение элементов, изменение сигнатур методов, переименование и изменение типов переменных, а также переименование полей.

Далеко не все инструменты рефакторинга и трансформации ориентированы непосредственно на исходный код. Например, в работе [54] описан инструмент ASM, который позволяет манипулировать и трансформировать классы в бинарных файлах с байткодом Java.

Также, существуют инструменты, предназначенные исключительно для поиска и сопоставления языковых конструкций без применения замен или трансформаций. К таким инструментам относятся ast-grep [55] и Semgrep [56]. Оба инструмента поддерживают множество языков и осуществляют сопоставление программного кода на уровне AST. Кроме того, язык запросов CodeQL [57] и его инструментарий позволяют выполнять поиск и сопоставление конструкций в программном коде с использованием декларативных запросов без его непосредственной замены.

3.2 Обзор исследований

В данном подразделе представлен обзор исследований на тему миграции API, описывающих проблемы, возникающие как при определении необходимости миграции, так и при осуществлении самой миграции. Рассматриваемые результаты исследований не нашли практическое применение в виде полноценных инструментов (не прототипов) или программных решений, соответственно нет возможности полноценно протестировать их для оценки возможности осуществления определенных классов миграций API.

3.2.1 Исследование практического применения методов автоматизированного переноса API Android

В работе [58] авторы обсуждают свой опыт миграции Android API, включая подробное рассмотрение документации. Основное внимание уделяется выявлению проблем,

возникающих при миграции программного кода, а также определению случаев, когда миграция оказывается необходимой.

Авторы выделяют несколько проблем, связанных с миграцией API, которые рассматриваются как неразрешимые:

1. Проблема миграция методов API «N-to-N», которая выражается в необходимости поиска соответствия методов из разных наборов API друг другу.
2. Проблема определения ложноположительных случаев среди множества возможных миграций.
3. Проблема определения временных интервалов между необходимостью добавления и удаления API.
4. Проблема определения миграции кода (значительное количество (около 66,3%) исследованных модификаций API не содержало миграций кода, то есть замены отсутствовали или находились в труднодоступном для поиска состоянии).

Также в работе представлены способы поиска информации об истории миграции API в проектах и описаны основные проблемы, которые могут возникнуть во время этого поиска.

3.2.2 Полуавтоматическое обновление приложений в ответ на изменения API библиотеки

В статье [59] предлагается метод распространения правил миграции кода через заголовочные файлы библиотеки. Правила интегрируются в грамматику языка, используемого для реализации приложения, что требует генерации нового синтаксического анализатора. Миграция API выполняется на уровне абстрактного синтаксического дерева (AST) с использованием инструментов PCCTS (Purdue Compiler Construction Tool Set) [60] и Sorcerer [61]. Текущая реализация данного подхода представлена в виде прототипа инструмента для обновления кода на языке C, который способен:

1. Добавлять и удалять включаемые файлы или изменять их имена;
2. Добавлять, удалять или переименовывать имена функций;
3. Перемещать функции между включаемыми файлами;
4. Добавлять, удалять и переупорядочивать параметры функций;
5. Модифицировать аргументы по умолчанию;
6. Изменять типы возвращаемых значений функций;
7. Изменять типы параметров функций;
8. Изменять значения входящих параметров функций;
9. Удалять, добавлять, переименовывать и изменять поля структур;
10. Удалять, добавлять, переименовывать и изменять глобальные переменные.

Статья также представляет низкоуровневый язык AST-DSL, позволяющий описывать миграции API в виде набора правил для разработанного прототипа инструмента. Например, правило для миграции сигнатуры функции от `assign(char*, char*, int=NPOS)` к `assign(char*, char*, int=0, int=NPOS)` представлено на листинге 32.

В поле `FNAME` указана функция, которая была изменена разработчиком. А в поле `PATTERN` задана синтаксическая структура вызова функции, которая может использоваться для сопоставления с любым вызовом. Если фрагмент целевого кода сопоставим с полем `PATTERN`, инструмент выполняет код, указанный в поле `ACTION`. В данном случае добавляется четвертый аргумент к функции со значением по умолчанию равным 0.

Разработанный прототип инструмента далек от завершения и способен выполнять лишь простейшие миграции программного кода, что ограничивает его использование для высокоуровневых миграций.

```
/*
* _ASE_BEGIN_
* FNAME = assign
* PATTERN = #(fc:FCALL_assign id:ID lp:L_PAREN arg1:gen_expr
*           co1:COMMA arg2:gen_expr co2:COMMA arg3:gen_expr rp:R_PAREN)
* ACTION = #gen_logical_expr =
*           #(fc, #id, #lp, #arg1, #co, #arg2, #co, 0, #co2, #arg3, #rp);
* _ASE_END_
*/
```

Листинг 32. Правило трансформации на AST-DSL.
Listing 32. Transformation rule in AST-DSL.

3.2.3 Swing to SWT и обратно: шаблоны для миграции API с помощью переноса программного кода

В работе [62] рассматривается миграция между платформами Swing [63] и SWT Java GUI [64] с применением шаблона «Адаптер» [65]. Авторы выделяют несколько основных проблем, возникающих при адаптации одного API к другому:

1. Нетривиальность отображения методов (например, «один ко многим» и «многие ко многим»);
2. Различия в иерархиях классов;
3. Поиск соответствия между идентификаторами объектов;
4. Проблемы использования нового API целевой платформой.

Авторы приводят описание потенциальной реализации инструмента автоматической миграции, который позволяет создавать адаптационные оболочки для старого API в целях совместимости с новым.

3.2.4 Автоматизированное обновление API для приложений Android

В работе [66] представлен метод AppEvolve, предназначенная для автоматических обновлений использования Android API. Метод заключается в обновлении использования Android API в коде целевого приложения, путем заимствования опыта миграции программного кода других разработчиков на новые версии API. Авторы выделяют 4 основных этапа обновления приложения под новую версию Android API:

1. Анализ использования API: на этом этапе выполняется анализ исходного кода приложения с целью определения мест, требующих изменения.
2. Поиск примеров обновлений: здесь информация с первого этапа соединяется со спецификацией изменений API для поиска актуальных примеров обновлений.
3. Обновление примеров анализа: найденные примеры миграции API обрабатываются, обобщаются и ранжируются по степени соответствия нуждам пользователя.
4. Обновление использования API: на этом этапе используются примеры, отобранные на предыдущем этапе, для внесения обновлений в API, а также производится тестирование обновленного кода с помощью дифференциального тестирования.

Результаты, полученные в ходе исследования, носят теоретический характер и не нашли полноценное применения в программном решении обновления приложений при миграции Android API.

3.2.5 Влияние переноса API на качество и понимание программного обеспечения

В работе [67] исследуется влияние переноса API на качество и понимание программного обеспечения. Подход, разработанный авторами, основывается на:

- Правилах адаптации программного кода;
- Методах отображения старого API на новый;
- Адаптационном рефакторинге;
- Принципе связности;
- Принципе сплоченности.

Правила адаптации программного кода содержат описание пар «Источник» – «Цель», где «Источник» представляет исходную библиотеку или класс, а «Цель» – целевую библиотеку или класс, к которым требуется адаптация. Например, правило `ClassSrc – ClassDst` означает переноса класса `ClassSrc` в новый класс `ClassDst`.

Методы отображения определяются как процесс замены одного или нескольких классов или методов из исходной библиотеки классами целевой библиотеки.

Рефакторинг определяется как процесс изменения программного приложения, который улучшает качество приложения, не изменяя его поведения [68-69]. Рефакторинг является одним из наиболее распространенных методов повышения качества программного приложения [69-70]. Существуют различные операции рефакторинга, которые могут быть использованы для улучшения качества программного приложения, включая изменение типов параметров, перемещение атрибутов и методов, переименование переменных, извлечение методов и классов и другие.

Принцип связности подразумевает измерение уровня взаимосвязи между модулями программного приложения, где желательной является слабая связь (то есть меньшая зависимость между модулями) [71]. Для вычисления этой метрики можно использовать связь между объектами (Coupling Between Objects, CBO), соединённую с модулями. Чем выше CBO, тем выше класс связности приложения.

Принцип сплоченности заключается в необходимости измерения взаимосвязей внутри отдельных модулей программного приложения. Сильное взаимодействие между элементами повышает удобство сопровождения кода [71]. Для оценки сплоченности классов достаточно использовать одну метрику – нормализованное отсутствие сплоченности методов (Normalized Lack of Cohesion of Methods, LCOM) [72-73]. Метрика LCOM показывает, насколько методы связаны друг с другом через атрибуты. Если все методы обращаются к одинаковым атрибутам, значение LCOM равно 0, что свидетельствует о высокой связности методов внутри класса, в противном случае высокое значение LCOM указывает на слабую связь, что увеличивает вероятность ошибок при проектировании и разработке программного приложения.

Результаты, полученные в ходе исследования, носят теоретический характер и не нашли своего применения в полноценном инструменте обновления приложений при миграции API.

3.2.6 Адаптация программных приложений к новым версиям API от компании Google

Работа [74] содержит рекомендации от Google, направленные на перенос программного кода приложений, написанных под операционную систему Android, с устаревших версий API на более новые. Авторы охватывают ключевые изменения в API, а также новые функции и улучшения. Однако эти рекомендации касаются только ручной адаптации кода под новые версии API.

3.2.7 Другие исследования

Существует ряд статистических исследований, подчеркивающих актуальность инструментов миграции API и опыт их использования. Однако воспроизведение их результатов зачастую бывает затруднительным.

В работе [75] рассматриваются аспекты устаревшего Android API, в то время как работа [76] сосредоточена на устаревании API платформы Java Development Kit. Исследование [77] охватывает общие проблемы, связанные с API, включая масштаб изменений, время реагирования на изменения, и способы, которыми системы справляются с изменениями API. В работе [78] представлен обзор стабильности версий API и особенностей миграции в экосистеме Android, тогда как работа [79] исследует возможности миграции программного кода между языками программирования C# и Java. В работе [80] описывается метод миграции между реализациями функционально схожих библиотек для разных программно-аппаратных платформ с использованием языка программных аннотаций (Program Annotation Language – PanLang) собственной разработки. С помощью данного языка можно описать каждый функциональный элемент каждой библиотеки (например, глобальные переменные и функции), чтобы затем сопоставить их по семантическому смыслу при миграции программного кода, который их использует. В качестве примера авторы описывают миграцию кода, использующего сетевую библиотеку BSD сокетов в код, использующий библиотеку Winsock. Работа [81] содержит статистические данные о деструктивных изменениях API.

Также проводятся исследования трансформаций программного кода с использованием методов искусственного интеллекта и машинного обучения, разбор которых выходит за рамки данной работы – [82] и [83].

4. Выводы

Применимость рассмотренных инструментов обусловлена сложностью реализованных в них подходов. Каждый из рассмотренных инструментов будет полезен при осуществлении миграций, принадлежащих определенным классам из раздела 2.

Например, такие инструменты как *ast-grep* (см. раздел 3.1.16), *Comby* (см. раздел 3.1.9) или *GoPatch* (см. раздел 3.1.11) предоставляют функциональные возможности для простого поиска и замены языковых конструкций (выражений или отдельных операторов) как по исходному коду, так и по его внутреннему представлению (например, AST) без проведения семантических проверок. Эти инструменты будут полезны при решении простых задач по миграции или трансформации кода, таких как, переименование переменной или функции, или замены сигнатуры метода или функции. Таким образом, с помощью данных инструментов возможно производить автоматические миграции программного кода первого, третьего, четвертого и пятого подклассов низкоуровневого класса миграции API из таблицы 1 (см. раздел 2.6) для кодовых баз различного размера.

Инструменты, предоставляющие возможности для семантического анализа конструкций языка программирования перед их сопоставлением и трансформацией, обычно обладают DSL собственного проектирования и разработки, например, инструменты *Coccinelle* (см. раздел 3.1.1), *Coccinelle4J* (см. раздел 3.1.1) и язык *PATL* (см. раздел 3.1.2), или предоставляют библиотеку и инструментарий для этого, например, *SPOON* (см. раздел 3.1.8) или *OpenRewrite* (см. раздел 3.1.10). С помощью инструментов данной категории возможно проводить большинство миграций программного кода первого, третьего, четвертого и пятого подклассов низкоуровневого класса миграции, а также определенные виды сложных миграций программного кода второго подкласса низкоуровневого класса миграции и высокоуровневого класса миграций из таблицы 1. Но, автоматическая трансформация программного кода произвольной сложности без верификации и поддержки разработчиков

не осуществима из-за ограничений данных инструментов. В частности, ряд рассматриваемых инструментов, таких как Coccinelle и Coccinelle4J, не учитывает поток управления программы или граф наследования классов. Кроме того, некоторые из этих решений не предлагают достаточно выразительные языки для описания сложных правил семантического преобразования, что требуется дополнительной квалификации конечного пользователя для использования полного набора их возможностей. Это относится, например, к решениям Nobrainer (см. раздел 3.1.4), Refaster (см. раздел 3.1.6), Twining (см. раздел 3.1.3), SWIN (см. раздел 3.1.3), Proteus (см. раздел 3.1.7), Stratego (см. раздел 3.1.7) и Rascal (см. раздел 3.1.15). Для различных задач миграции и трансформации программного кода, также может быть важно представление кода, с которым работает тот или иной инструмент. Не все анализируемые инструменты функционируют на уровне исходного кода программы. Например, инструмент ASM ориентирован на работу с байт-кодом, а не с исходным. В то же время Coccinelle предлагает интуитивно понятный для пользователя язык семантических преобразований SmPL, а Nobrainer реализует инновационный подход на основе механизма сопоставления абстрактного синтаксического дерева (AST) компилятора Clang.

Рассмотренные исследования на тему миграции API (подраздел 3.2) разделяются на работы, исследующие проблемы, связанные с миграцией API и работы, предлагающие методы решения проблемы миграции для конкретных случаев (например, миграция приложений при обновлении Android API (подраздел 3.2.4)). Общей характеристикой всех рассмотренных исследований является преобладание достигнутых авторами теоретических результатов над практическими.

5. Заключение

В ходе проведенного исследования были отобраны и классифицированы примеры миграций приложений на языке Java под различные версии Android API. По отобранным примерам было выделено 2 типовых класса миграции API – низкоуровневый и высокоуровневый. Низкоуровневый класс миграции API был дополнительно классифицирован на 5 подклассов. Выделенные классы миграции носят общий характер и не привязаны к миграции Android API или языку Java. Их можно обобщить на случай миграции произвольной библиотеки или произвольного объектно-ориентированного языка. Далее были рассмотрены программные решения, предназначенные для решения задач автоматического и автоматизированного переноса программного кода в процессе миграции API для различных языков программирования, включая C, C++, Java и JavaScript, а также исследования, посвященные проблеме миграции API. Критерии рассмотрения инструментов включали возможность осуществления миграций из выделенных классов и подклассов миграции API, а также удобство использования их конечным пользователем, исходя из предположения, что правила миграций предпочтительнее описывать на целевых языках или на схожих с ними DSL.

Подводя итог, важно подчеркнуть, что ни одно из рассмотренных программных средств не предоставляет возможности для простого и полнофункционального автоматического переноса произвольного программного кода на конкретном языке программирования с точки зрения примеров произвольной сложности из каждого выделенного класса миграции API. Это свидетельствует о необходимости дальнейших исследований и разработок в данной области, направленных на создание более универсальных и эффективных инструментов для автоматизации миграции кода.

В табл. 2 представлен сравнительный список рассмотренных программных инструментов миграции API.

Табл. 2. Сравнительный список программных инструментов для миграции API.
Table 2. Comparison List of API Migration Software Tools.

Инструментарий	Тип лицензии	Поддерживаемые языки	Комментарий
Coccinelle	открытая	C	поддерживает трансформации только в пределах отдельных функций / подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
Coccinelle4J	закрытая	Java	поддерживает трансформации только в пределах отдельных методов / подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
Nobrainier	закрытая	C/C++	поддерживает трансформацию ограниченного подмножества конструкций целевого языка / подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
MARTINI	открытая	C/C++	находится в разработке и поддерживает трансформацию ограниченного подмножества конструкций целевого языка/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
Refaster	открытая	Java	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
Proteus	закрытая	C/C++	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
SPOON	открытая	Java	поддерживает трансформацию ограниченного подмножества конструкций целевого языка / подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
Comby	открытая	мультиязычный	пригоден лишь для проведения простых трансформаций/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
OpenRewrite	открытая	Java	сложность описания правил трансформации/подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
GoPatch	открытая	Go	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
CodeTurn	закрытая	COBOL	поддерживает трансформацию ограниченного подмножества конструкций целевого языка/подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
Rascal	открытая	Java	отсутствуют семантические проверки при сопоставлении/подходит для осуществления большинства миграций, принадлежащих низкоуровневому классу миграций
JSCodeshift	открытая	JavaScript	отсутствуют семантические проверки при сопоставлении/подходит для осуществления миграций из первого, третьего и четвертого подклассов низкоуровневого класса миграций
ClangMR	открытая	C/C++	требуются знания API Clang для проведения сложных трансформаций/ подходит для осуществления большинства миграций, принадлежащих низкоуровневую классу, а также с помощью него возможно осуществлять высокоуровневые миграции

Список литературы / References

- [1]. What is an API? (Application Programming Interface). <https://www.mulesoft.com/resources/api/what-is-an-api> (accessed 01.09.2024).
- [2]. CMake Documentation and Community. <https://cmake.org/documentation/> (accessed 01.09.2024).
- [3]. Android API levels. <https://apilevels.com> (accessed 01.09.2024).
- [4]. Lamothe M., Shang W., Chen T. H. P. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering*, 2020, vol. 48, issue 2, pp. 417-431, DOI: 10.1109/TSE.2020.2988396.
- [5]. Top Programming Languages for Android App Development in 2023. <https://medium.com/agileinsider/top-programming-languages-for-android-app-development-in-2023-3eb5d> (accessed 01.09.2024).
- [6]. Github. <https://github.com/> (accessed 01.09.2024).
- [7]. Rodriguez L. R. and Lawall J. Increasing Automation in the Backporting of Linux Drivers Using Coccinelle. In 11th European Dependable Computing Conference (EDCC), Paris, France, 2015, pp. 132-143, DOI: 10.1109/EDCC.2015.23.
- [8]. Kang H. J., Thung F., Lawall J., Muller G., Jiang L., Lo D. Semantic Patches for Java Program Transformation (Experience Report). In 33rd European Conference on Object-Oriented Programming (ECOOP 2019). *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 134, pp. 22:1-22:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2019), DOI: 10.4230/LIPIcs.ECOOP.2019.22.
- [9]. Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. Transforming Programs between APIs with Many-to-Many Mappings. In 30th European Conference on Object-Oriented Programming (ECOOP 2016). *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 56, pp. 25:1-25:26, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2016), DOI: 10.4230/LIPIcs.ECOOP.2016.25.
- [10]. M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 2010, pp. 205-214, DOI: 10.1145/1806799.1806832.
- [11]. Jun Li, Chenglong Wang, Yingfei Xiong, Zhenjiang Hu. SWIN: Towards Type-Safe Java Program Adaptation between APIs. *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, 2015, pp. 91-102, DOI: 10.1145/2678015.2682534.
- [12]. Savchenko V. V., Sorokin K. S., Pankratenko G. A., Markov S., Spiridonov A., Alexandrov I., Volkov A. S., and Sun, K.-W. Nobrainer: An example-driven framework for C/C++ code transformations. *Perspectives of System Informatics: 12th International Andrei P. Ershov Informatics Conference, PSI 2019, Novosibirsk, Russia, July 2–5, 2019, Revised Selected Papers 12*, Springer International Publishing, 2019, pp. 140-155, DOI: 10.1007/978-3-030-37487-7_12.
- [13]. Clang: a C language family frontend for LLVM. <https://clang.llvm.org/> (accessed 01.09.2024).
- [14]. The LLVM Compiler Infrastructure. <https://llvm.org/> (accessed 01.09.2024).
- [15]. AST Matcher Reference. <https://clang.llvm.org/docs/LibASTMatchersReference.html> (accessed 01.09.2024).
- [16]. The Official YAML Web Site. <https://yaml.org/> (accessed 01.09.2024).
- [17]. Johnson A., Coti C., Malony A.D., Doerfert J. MARTINI: The Little Match and Replace Tool for Automatic Application Rewriting with Code Examples. In: Cano, J., Trinder, P. (eds) *Euro-Par 2022: Parallel Processing. Euro-Par 2022. Lecture Notes in Computer Science*, vol 13440. Springer, Cham., 2022, vol. 7, issue 76, pp. 4590, DOI: 10.1007/978-3-031-12597-3_2.
- [18]. Wasserman L. Scalable, example-based refactorings with refaster. *WRT'13: Proceedings of the 2013 ACM Workshop on refactoring tools*, pp. 25-28, DOI: 10.1145/2541348.2541355.
- [19]. Waddington D., Yao B. High-fidelity C/C++ code transformation. *Science of Computer Programming*, 2007, vol. 68, issue 2, pp. 64-78, DOI: 10.1016/j.scico.2006.04.010.
- [20]. Visser E. (2001). *Stratego: A Language for Program Transformation Based on Rewriting Strategies* System Description of Stratego 0.5, 2001, pp. 357-361. Springer, Berlin, Heidelberg. DOI: 10.1007/3-540-45127-7_27.
- [21]. Pawlak R. et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 2016, vol. 46, issue 9, pp. 1155-1179, DOI: 10.1002/spe.2346.
- [22]. Long F., Amidon P., Rinard M. Automatic inference of code transforms for patch generation. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727-739, DOI: 10.1145/3106237.3106253.

- [23]. Saha R. K., Lyu Y., Yoshida H. and Prasad M. R. Elixir: Effective object-oriented program repair, 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 2017, pp. 648-659, DOI: 10.1109/ASE.2017.8115675.
- [24]. Wen M., Chen J., Wu R., Hao D. and Cheung S.-C., Context-Aware Patch Generation for Better Automated Program Repair, 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 2018, pp. 1-11, DOI: 10.1145/3180155.3180233.
- [25]. Xuan J., Xie X., Monperrus M. Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help. ESEC/FSE 2015 - 10th Joint Meeting on Foundations of Software Engineering, NIER Track, Aug 2015, Bergamo, Italy. pp.910-913, DOI: 10.1145/2786805.2803206.
- [26]. Xuan J. et al. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. IEEE Transactions on Software Engineering, 2016, vol. 43, issue 1, pp. 34-55, DOI: 10.1109/TSE.2016.2560811.
- [27]. Martinez M., Monperrus M. Astor: A program repair library for java. Proceedings of the 25th international symposium on software testing and analysis, 2016, pp. 441-444, DOI: 10.1145/2931037.2948705.
- [28]. Van Tonder R., Le Goues C. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019, pp. 363-378, DOI: 10.1145/3314221.3314589.
- [29]. OpenRewrite: Semantic code search and transformation tool. Moderne Inc., <https://docs.openrewrite.org/> (accessed 01.09.2024).
- [30]. Introduction to Apache HttpClient. <https://blog.knoldus.com/introduction-to-apache-httpclient/> (accessed 01.09.2024).
- [31]. Lossless Semantic Trees (LST). <https://docs.openrewrite.org/concepts-explanations/lossless-semantic-trees> (accessed 01.09.2024).
- [32]. go-patch: Structured code diffs and refactors. Uber Technologies, Inc. GitHub repository. <https://github.com/uber-go/gopatch> (accessed 01.09.2024).
- [33]. Detailed Description of Unified Format. https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html (accessed 01.09.2024).
- [34]. CodeTurn: Automated Mainframe Code Conversion. <https://www.astadia.com/products/codeturn> (accessed 01.09.2024).
- [35]. Cordy J. R. The TXL source transformation language. Science of Computer Programming, 2006, vol. 61, issue 3, pp. 190-210, DOI: 10.1016/j.scico.2006.04.002.
- [36]. Jscodeshift. <https://github.com/facebook/jscodeshift> (accessed 01.09.2024).
- [37]. Recast. <https://github.com/benjamn/recast> (accessed 01.09.2024).
- [38]. AST-types. <https://github.com/benjamn/ast-types> (accessed 01.09.2024).
- [39]. AVA. <https://github.com/avajs/ava> (accessed 01.09.2024).
- [40]. React-codemod. <https://github.com/reactjs/react-codemod> (accessed 01.09.2024).
- [41]. Lodash-codemods. <https://github.com/jfmengels/lodash-codemods> (accessed 01.09.2024).
- [42]. ESTree. <https://github.com/estree/estree> (accessed 01.09.2024).
- [43]. Using the Compiler API. <https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API> (accessed 01.09.2024).
- [44]. Klint P., van der Storm T., Vinju J. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, Edmonton, AB, Canada, 2009, pp. 168-177, DOI: 10.1109/SCAM.2009.28.
- [45]. Wright H. K., Jasper D., Klimek M., Carruth C. and Wan Z. Large-Scale Automated Refactoring Using ClangMR, 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 2013, pp. 548-551, DOI: 10.1109/ICSM.2013.93.
- [46]. Koppel J., Premtoon V., Solar-Lezama A. One tool, many languages: language-parametric transformation with incremental parametric syntax. Proceedings of the ACM on Programming Languages, 2018, vol. 2, issue OOPSLA, pp. 1-28, DOI: 10.1145/3276492.
- [47]. Ketkar A. et al. A Lightweight Polyglot Code Transformation Language. Proceedings of the ACM on Programming Languages, 2024, vol. 8, issue PLDI, article No.: 199, pp. 1288-1312, DOI: 10.1145/3656429.
- [48]. Max Brunsfeld Tree-sitter. GitHub. <https://tree-sitter.github.io/tree-sitter/> (accessed 01.09.2024).
- [49]. Ramanathan M. K., Clapp L., Barik R. and Sridharan M. Piranha: Reducing Feature Flag Debt at Uber, 2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Seoul, Korea (South), 2020, pp. 221-230, DOI: 10.1145/3377813.3381350.

- [50]. Baxter I. D. DMS: Program transformations for practical scalable software evolution. IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, pp. 48-51, DOI: 10.1145/512035.512047.
- [51]. Dagenais B., Robillard M. P. SemDiff: Analysis and recommendation support for API evolution. 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 2009, pp. 599-602, DOI: 10.1109/ICSE.2009.5070565.
- [52]. Henkel J., Diwan A. CatchUp! Capturing and replaying refactorings to support API evolution. Proceedings, 27th International Conference on Software Engineering, 2005. ICSE 2005., St. Louis, MO, USA, 2005, pp. 274-283, DOI: 10.1109/ICSE.2005.1553570.
- [53]. Eclipse. <https://www.eclipse.org/> (accessed 01.09.2024).
- [54]. Bruneton E., Lenglet R., Coupaye T. ASM: a code manipulation tool to implement adaptable systems. In Adaptable and extensible component systems, 2002, vol. 30, issue 19.
- [55]. ast-grep: write code to match code. Open-source. <https://ast-grep.github.io> (accessed 01.09.2024).
- [56]. Semgrep. Open-source. <https://github.com/semgrep/semgrep> (accessed 01.09.2024).
- [57]. CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security. GitHub, Inc., <https://codeql.github.com> (accessed 01.09.2024).
- [58]. Lamothe M., Shang W. Exploring the use of automated API migrating techniques in practice: an experience report on android. Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 503-514, DOI: 10.1145/3196398.3196420.
- [59]. Chow, Notkin. Semi-automatic update of applications in response to library changes. 1996 Proceedings of International Conference on Software Maintenance, Monterey, CA, USA, 1996, pp. 359-368, DOI: 10.1109/ICSM.1996.565039.
- [60]. Parr T. J., Dietz H. G., Cohen W. E. PCCTS reference manual: version 1.00. ACM SIGPLAN Notices, 1992, vol. 27, issue 2, pp. 88-165, DOI: 10.1145/130973.130980.
- [61]. Parr T. J. An Overview of SORCERER: A Simple Tree-Parser Generator. Poster paper; International Conference on Compiler Construction 1994; Edinburgh, Scotland; April 1994.
- [62]. Bartolomei T. T., Czarnecki K., Lämmel R. Swing to SWT and back: Patterns for API migration by wrapping. 2010 IEEE International Conference on Software Maintenance, Timisoara, Romania, 2010, pp. 1-10, DOI: 10.1109/ICSM.2010.5610429.
- [63]. Java Swing Tutorial. <https://www.javatpoint.com/java-swing> (accessed 01.09.2024).
- [64]. SWT: The Standard Widget Toolkit. <https://www.eclipse.org/swt/> (accessed 01.09.2024).
- [65]. Adapter Pattern. <https://www.geeksforgeeks.org/adapter-pattern/> (accessed 01.09.2024).
- [66]. Fazzini M., Xin Q., Orso A. Automated API-usage update for Android apps. ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 204-215, DOI: 10.1145/3293882.3330571.
- [67]. Alrubaye H., Alshoaibi D., Alomar E., Mkaouer M. W., and Ouni A. How Does API Migration Impact Software Quality and Comprehension? An Empirical Study. 2019, DOI: 10.48550/arXiv.1907.07724.
- [68]. Opdyke W. F. Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign, 1992.
- [69]. Fowler M. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.
- [70]. Stroggylos K., Spinellis D. Refactoring – Does It Improve Software Quality? Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007), Minneapolis, MN, USA, 2007, pp. 10-10, DOI: 10.1109/WOSQ.2007.11.
- [71]. Stevens W. P., Myers G. J., Constantine L. L. Structured design. In IBM Systems Journal, vol. 13, issue 2, pp. 115-139, 1974, DOI: 10.1147/sj.132.0115.
- [72]. Pantiuchina J., Lanza M., Bavota G. Improving Code: The (Mis) Perception of Quality Metrics, 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 80-91, DOI: 10.1109/ICSME.2018.00017.
- [73]. Chávez A., Ferreira I., Fernandes E., Cedrim D., Garcia A. SBES '17: Proceedings of the XXXI Brazilian Symposium on Software Engineering, pp. 74-83, DOI: 10.1145/3131151.3131171.
- [74]. Migrating your apps to Android 9. <https://developer.android.com/about/versions/pie/android-9.0-migration> (accessed 01.09.2024).
- [75]. Li L. et al. Characterising deprecated Android APIs. MSR '18: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 254 – 264, DOI: 10.1145/3196398.3196419.
- [76]. Sawant A.A., Robbes R. & Bacchelli A. On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK. Empirical Software Engineering, vol. 23, issue 4, 2158–2197 (2018). DOI: 10.1007/s10664-017-9554-9.

- [77]. Hora A., Robbes R., Anquetil N., Etien A., Ducasse S., Valente M. T. How Do Developers React to API Evolution? The Pharo Ecosystem Case, 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 2015, pp. 251-260, DOI: 10.1109/ICSME.2015.7332471.
- [78]. Itsykson V., Zozulya A. Automated program transformation for migration to new libraries, 2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR), Moscow, Russia, 2011, pp. 1-7, DOI: 10.1109/CEE-SECR.2011.6188463.
- [79]. Brito A., Xavier L., Hora A., Valente M. T. Why and how Java developers break APIs, 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 2018, pp. 255-265, DOI: 10.1109/SANER.2018.8330214.
- [80]. McDonnell T., Ray B., Kim M. An Empirical Study of API Stability and Adoption in the Android Ecosystem. 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 2013, pp. 70-79, DOI: 10.1109/ICSM.2013.18.
- [81]. Pandita R., Jetley R. P., Sudarsan S. D., Williams L. Discovering likely mappings between APIs using text mining. 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), Bremen, Germany, 2015, pp. 231-240, DOI: 10.1109/SCAM.2015.7335419.
- [82]. Li Y., Wang S., Nguyen T. N. DLFix: Context-based code transformation learning for automated program repair. ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 602 – 614, DOI: 10.1145/3377811.3380345.
- [83]. Thongtanunam P., Pornprasit C., Tantithamthavorn C. Autotransform: Automated Code Transformation to Support Modern Code Review Process. ICSE '22: Proceedings of the 44th International Conference on Software Engineering, pp. 237 – 248, DOI: 10.1145/3510003.3510067.

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.

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

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Dmitry Mikhailovich MELNIK – Senior Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

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 in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

DOI: 10.15514/ISPRAS-2025-37(4)-8



Реализация гибкой системы управления доступом в приложении

Т.И. Васильев, ORCID: 0009-0008-6671-3026 <timurvasilev@gmail.com>

Paybis USA LTD, США, Талса, Бостон Авеню 321.

Аннотация. В статье рассматривается развитие механизма конфигурации гибкой ролевой системы для приложений, которые требуют динамического разграничения прав пользователей в зависимости от бизнес-контекста или различных условий. Современные приложения становятся все больше и сложнее, что приводит к необходимости реализации эффективных механизмов контроля доступа. Описанный подход предполагает использование собственной ролевой модели, которая позволит точно настраивать права пользователей для различных сущностей в базе данных. Основное внимание уделено гибкости и масштабируемости модели, возможности настраивать доступ в зависимости от статусов объектов, ролей пользователей и нужд бизнеса. В качестве примера рассмотрена конфигурация интернет-магазина, с разграничением по ролям пользователей и учетом различий в статусах товара. Подробно описан процесс настройки доступов, приведён пример кода для проверки разрешений и динамического отображения полей. В заключение обсуждаются преимущества данного механизма и его недостатки.

Ключевые слова: гибкая система управления доступом; ролевая модель; контроль доступа; права пользователей; информационная безопасность; JSON-конфигурации; интернет-магазин; управление доступом; разграничение прав доступа; масштабируемость информационных систем.

Для цитирования: Васильев Т.И. Реализация гибкой системы управления доступом в приложении. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 147–160. DOI: 10.15514/ISPRAS-2025-37(4)-8.

Implementation of a Flexible Access Control System in the Application

T.I. Vasilev, ORCID: 0009-0008-6671-3026 <timurvasilev@gmail.com>

Paybis USA LTD, USA, Tulsa, 321 S Boston Ave Downtown.

Abstract. The article considers the development of a flexible role system configuration mechanism for applications that require dynamic differentiation of user rights depending on the business context or various conditions. Modern applications are becoming more and more complex, which leads to the need to implement effective access control mechanisms. The described approach involves the use of its own role model, which will allow you to fine-tune user rights for various entities in the database. The focus is on the flexibility and scalability of the model, which allows you to customize access depending on the statuses of objects, user roles and business needs. As an example, the configuration of an online store is considered, with differentiation by user roles and various product statuses. The access configuration process is also described in detail and an example code is provided for checking permissions and dynamically displaying fields. In conclusion, the advantages of this mechanism and its disadvantages are discussed.

Keywords: flexible access control system; role model; access control; user rights; information security; JSON configurations; online store; access control; access rights differentiation; scalability of information systems.

For citation: Vasilev T.I. Implementation of a flexible access control system in the application, vol. 37, issue 4, part 1, 2025, pp. 147-160 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-8.

1. Введение

В современном мире информационные системы становятся все более объёмными и сложными, требуя надёжного и гибкого подхода к управлению доступом. Важным аспектом безопасности данных в веб-приложениях является контроль над тем, какие пользователи системы могут обращаться к той или иной информации или имеют право на выполнение того или иного действия. Для достижения наибольшей гибкости и удобства системы в управлении правами доступа возникает необходимость построения собственного механизма ролевой системы, с помощью которой будет возможно конфигурировать права пользователей к разным аспектам системы в зависимости от текущих условий и контекста использования. Данный механизм должен быть масштабируем и легко применим в возможных сценариях работы системы без значительных архитектурных изменений в приложениях. В статье описывается подход к реализации системы управления доступом, основанной на гибкой ролевой модели в веб-приложениях с использованием реляционной базы данных MySQL или Postgres [1]. Предложенный механизм был применён в нескольких проектах и продемонстрировал свою эффективность в разнообразных сценариях. Целью работы является представление разработанного механизма, а также анализ его возможностей в сравнении с альтернативными решениями. Основной целью проведенной работы по адаптации известных подходов являлось повышение их гибкости, что особенно актуально для веб-приложений, работающих с реляционными базами данных.

2. Обзор существующих решений

Устоявшимися и часто используемыми ролевыми моделями доступа в веб-приложениях являются модели RBAC и ABAC [2-3]. Ролевая модель доступа RBAC основывается на присваивании пользователям ролей с фиксированным набором прав. Положительными чертами данного подхода являются:

- Простота реализации RBAC позволяет быстро установить роли и назначить их пользователям, что существенно упрощает администрирование.
- Предсказуемость поведения системы, так как количество ролей и их комбинация ограничены.

- Простое управление доступом для больших групп пользователей за счёт предоставления или ограничения доступа на основе роли.

В качестве минусов можно выделить:

- Ограничение гибкости системы. Для роли фиксируется набор прав, который часто может не соответствовать набору необходимых разрешений для работы в системе.
- Сложность при изменении ролей. В больших системах с множеством ролей и пользователей достаточно сложно управлять изменениями ролевой системы.

Модель управления доступом ABAC использует атрибуты пользователей и объектов для динамического принятия решений о доступе. Основным отличием этого подхода в том, что каждая ситуация оценивается не с точки зрения роли пользователя и действия, которое он хочет совершить, а с точки зрения атрибутов, которые к ним относятся. Подробнее с данным механизмом можно ознакомиться в работе [4].

У модели ABAC имеются несколько достоинств:

- Данная модель позволяет динамично изменять политики доступа, что делает систему гибкой и масштабируемой.
- Благодаря тонкой настройке прав, доступ может быть настроен с учётом различных атрибутов.
- Модель ABAC особенно полезна в сценариях, где необходимо учитывать контекст запроса, такие как географическое местоположение или тип устройства.

В качестве недостатков можно выделить следующие:

- Сложность реализации модели и ее интеграции в веб-приложение. Требуется тщательно настроить правила для различных атрибутов и объектов, что значительно усложняет поддержку и разработку.
- При большом количестве атрибутов и правил проверка прав может стать достаточно медленной.

Подход, представленный в данной статье, позволяет более гибко настраивать доступ, но при этом сохраняет простоту и улучшает производительность по сравнению с моделями ABAC и RBAC.

К настоящему времени опубликовано множество научных работ по созданию ролевых моделей в различных программных системах. Исследование [5] предлагает анализ различных решений для построения ролевой модели управления доступом в корпоративных информационных системах. В работе рассмотрены как классические подходы к RBAC, так и их адаптации для сложных корпоративных сред. Ключевым преимуществом данной работы является структурированный подход к внедрению ролевой модели с учётом корпоративных требований. Однако в работе отсутствует описание практической реализации и примеры конкретных сценариев использования, что делает её сложной для использования в реальных проектах. В публикации [6] акцент сделан на использовании ролевой модели управления доступом в специализированной системе – системе конференций. Основное внимание уделяется тому, как модель RBAC может быть адаптирована для управления большими потоками участников, модераторов и организаторов. Положительной стороной является описание специфики доступа в контексте временных событий, таких как конференции. Однако данный подход сложно адаптировать для систем, не связанных с временными процессами, и отсутствуют универсальные рекомендации. Исследование [7] рассматривает применение модели RBAC в системах архивирования данных. Положительной стороной является учёт специфики работы с архивами, включая долгосрочное хранение и доступ к историческим данным. Недостатком является ограниченность подхода для использования в системах с более динамичным характером данных, например, в веб-приложениях. В работе [8] описана реализация гибкой системы управления доступом с акцентом на роли и возможности настройки уровня доступа в информационных системах. Основное

преимущество работы заключается в детализированном описании механизма доступа, который ориентирован на адаптивность под различные сценарии. Однако в работе отсутствует конкретика в применении технологий – отсутствует привязка к реальным инструментам или программным средствам, что может затруднить практическую реализацию.

Рассмотренные работы представляют различные подходы к реализации систем управления доступом. Тем не менее каждый из них имеет свои ограничения. Предложенное решение сочетает лучшие черты существующих подходов и предлагает гибкость, масштабируемость и простоту настройки, что делает его универсальным инструментом для современных веб-приложений.

3. Построение гибкой ролевой системы

Рассмотрим реализацию механизма на примере упрощённой версии интернет-магазина. Проектируемый интернет-магазин будет иметь дело со следующими объектами: товар и пользователь, остальные объекты, которых в реальной жизни много, в статье намерено опущены для упрощения представления реализации. Для хранения данных интернет-магазин использует реляционную SQL базу данных, что позволяет организовать объекты и их взаимосвязи в виде таблиц. В данном интернет-магазине будет использован следующий набор ролей:

- User – Роль для клиента интернет-магазина.
- Courier – Роль работника склада, осуществляющего доставку товара.

Товар будет иметь следующий набор статусов:

- Available – Доступен к заказу.
- NotAvailable – Закончился на складе и в данный момент недоступен.

При создании механизма сначала необходимо определить способ хранения конфигураций и значений параметров объектов. Суть механизма конфигураций в том, что у каждого типа объектов может существовать несколько описаний (конфигураций), где для каждого описания определяются наборы параметров и правила доступа к ним. Например, для типа «Товар» можно задать две разные конфигурации – «Книга» и «Телевизор». Конфигурация «Книга» включает, например, параметр «Автор» и «Количество страниц», а конфигурация «Телевизор» – «Диагональ экрана», «Разрешение» и другие характеристики.

Значения параметров объекта – это конкретные данные, которые присущи каждому экземпляру объекта. Например, один телевизор может иметь диагональ 40 дюймов, а другой – 50, но оба относятся к конфигурации «Телевизор» и типу «Товар». Аналогично, в случае «Книги» у отдельных экземпляров могут отличаться автор, количество страниц и другие характеристики.

Далее будут рассмотрены различные способы хранения конфигураций и значений параметров объектов, а также их преимущества и недостатки.

Первый способ хранения конфигураций изображен на рис. 1. Конфигурация представлена в виде JSON-файлов, которые хранятся в репозитории вместе с кодом приложения. Изменения в конфигурации выполняются программистами путем создания новых или редактирования существующих JSON-файлов с последующим подтверждением изменений в системе контроля версий.

Преимуществом такого подхода является возможность совместной работы команды программистов: изменения в конфигурации могут быть проверены через систему контроля версий, что обеспечивает их прозрачность и контроль качества. Кроме того, тестирование изменений на тестовых окружениях перед выпуском в рабочую среду помогает минимизировать риски, связанные с некорректной конфигурацией.

В процессе разработки и последующего использования системы, которая использовала данный подход к хранению конфигураций, могут возникать следующие проблемы:

- Скорость реагирования на сбои в рабочей среде может оказаться слишком низкой из-за того, что любые конфигурационные правки требуют полноценного развертывания, что во многих приложениях занимает существенное время.
- Заметная часть времени программистов может уходить на настройку конфигураций, а не на написание кода.

Таким образом, данный подход обеспечивает прозрачность и тестируемость изменений, но накладывает ограничения на оперативное управление конфигурациями в условиях реальной эксплуатации.

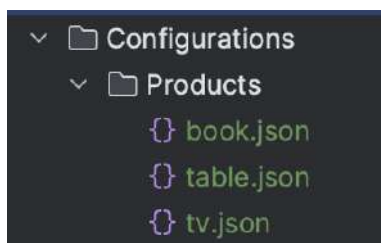


Рис. 1. Хранение конфигураций в коде приложения.
Fig. 1. Storing configurations in the application code.

Другим способом хранения конфигураций объектов является их размещение в базе данных приложения (рис. 2). В этом случае каждая конфигурация представлена отдельной записью в таблице. Данный подход даёт возможность разрабатывать административную панель, которая будет взаимодействовать непосредственно с базой данных и позволять вносить изменения в конфигурации через удобный пользовательский интерфейс, без участия программистов.

Чтобы понять разницу предложенных способов, рассмотрим общий процесс работы с конфигурациями при файловом хранении:

1. Программист изменяет конфигурационный файл в локальном репозитории и отправляет изменения в систему контроля версий.
2. Система сборки (CI/CD) при необходимости запускает тесты, формирует новую сборку приложения и подготавливает файлы для развёртывания.
3. Развёртывание в целевое окружение (тестовое, производственное и т.д.).

Если конфигурации размещены в файлах, чтобы внести даже незначительную правку (например, добавить новое поле или изменить права для роли пользователя), придётся снова проходить весь цикл: создать новую версию, выполнить сборку, а затем развернуть новые файлы на сервере. Возможен и упрощённый вариант действий, при котором в систему можно загрузить только изменённые файлы конфигурации без полноценной пересборки. Однако в этом случае пропускается этап автоматизированного тестирования, что увеличивает риск появления ошибок или несовместимых настроек, которые бы выявились при прохождении полного цикла сборки.

В случае же хранения конфигураций в базе данных вся цепочка может быть упрощена, если использовать административную панель:

1. Администратор с помощью веб-интерфейса получает доступ к списку существующих конфигураций.
2. Изменение конфигураций происходит путём прямого редактирования соответствующих записей в базе данных.
3. Сохранение правок сразу отражается в приложении, без пересборки и развёртывания всего проекта.

Configuration	
id	Int
configurations	Json
name	varchar(30)

Рис.2. Хранение конфигураций в базе данных.
Fig. 2. Storing configurations in a database.

Преимуществами данного подхода можно считать следующие:

- Администраторы, которые не обладают навыками программирования или не умеют работать с системой контроля версий, могут изменять настройки через удобный интерфейс, без необходимости редактирования JSON-файлов в коде приложения.
- Все изменения в конфигурациях могут быть применены оперативно через панель администрирования, что исключает необходимость пересборки приложения или развертывания.

Подход имеет и недостатки:

- Ручное внесение данных через веб-интерфейс повышает риск опечаток и неправильных значений, особенно если нет механизмов валидации или она недостаточно жёсткая. Такой риск может быть снижен продуманными строгими проверками на уровне приложения, однако полностью исключить человеческий фактор нельзя.
- Если изменения конфигурации вносятся индивидуально через панель, остальным участникам команды сложнее оперативно узнать, кто внес изменения и когда это произошло. В отличие от запросов (pull-request) или подтверждений (commit) в системе контроля версий, панель администрирования обычно не имеет встроенного механизма обсуждения и согласования правок.
- Создание и поддержка полноценной панели администрирования требует времени и ресурсов. Необходимо разрабатывать интерфейс, учитывать разные сценарии использования и права доступа, внедрять механизмы аудита изменений. Для небольших проектов это может оказаться избыточным.

Таким образом, переход на хранение конфигураций в базе данных и создание административной панели значительно упрощают процесс управления настройками приложения, но требуют внедрения дополнительных механизмов контроля и синхронизации данных для минимизации рисков.

Кроме традиционного хранения конфигураций в базе данных, можно использовать возможности PostgreSQL по работе с внешними таблицами (foreign tables). С помощью такого подхода данные могут фактически оставаться во внешнем источнике, но при этом выглядеть и обрабатываться в PostgreSQL, словно это обычные таблицы. Одним из примеров реализации этой концепции является расширение S3 FDW (Simple Storage Service Foreign-Data Wrapper), позволяющее хранить и читать данные из облачного сервиса Amazon S3 [9]. Это может быть полезно, если требуется гибкость и масштабируемость облачного хранилища, но при этом удобно пользоваться SQL-запросами и механизмами PostgreSQL для поиска, фильтрации и анализа данных. Такое решение частично примиряет идею «файлов в облаке» и «данных в СУБД», поскольку администратор может продолжать работать с конфигурациями через SQL-интерфейс, а сами файлы с настройками физически

размещаются на S3. В результате не требуется ручного копирования или синхронизации – система PostgreSQL сама обеспечивает связность и доступ к внешнему хранилищу.

В итоге, у каждого из рассмотренных подходов – будь то хранение конфигураций в файловой системе, в базе данных или с помощью внешних таблиц (к примеру, через S3 FDW) – есть свои преимущества и недостатки. Файлы проще в использовании и подходят для небольших проектов с редкими изменениями. Базы данных удобны для построения административных панелей, быстрой правки. Внешние таблицы совмещают гибкость базы данных и масштабируемость облачных хранилищ.

Выбор конкретного варианта зависит в первую очередь от потребностей проекта, сложности инфраструктуры, частоты и объёмов изменений, а также требований к безопасности и удобству эксплуатации. Лучше всего предварительно оценить все факторы и выбрать то решение, которое оптимально впишется в архитектуру и процессы команды.

Далее будут рассмотрены несколько подходов к реализации хранения параметров объекта.

На рис. 3 продемонстрирована схема базы данных с реализацией хранения значений параметров объекта в виде колонок в таблице. В данном подходе существенным минусом является необходимость изменения структуры таблицы базы данных при изменении конфигурации, когда при удалении или добавлении нового поля в конфигурации приходится также проводить миграцию данных с изменением структуры таблицы. Частота таких операций зависит от особенностей проекта, для систем с большим набором параметров объектов изменения могут быть ресурсоемкими и затруднять эксплуатацию. В небольших проектах с редкими изменениями конфигураций этот недостаток может оказаться незначительным.

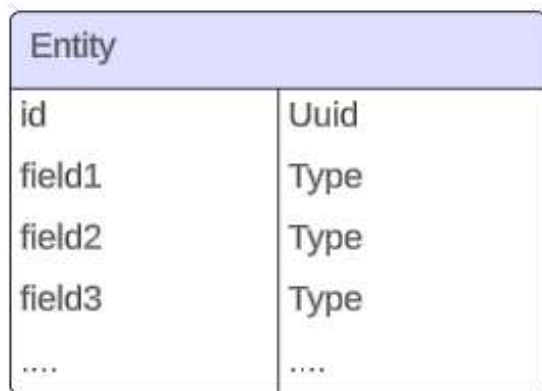


Рис. 3. Схема базы данных гибкой ролевой модели с сохранением данных в колонках.
Fig. 3. A flexible role model database schema with data stored in columns.

Другой способ реализации хранения значений параметров объекта для гибкой ролевой модели изображён на рис. 4, где каждое поле объекта вынесено как запись в отдельную таблицу. Используя данный подход, таблица с данными будет заполняться очень быстро, ведь на каждую запись в таблице Entity будет приходиться n записей (количество полей в наборе параметров объекта) в таблице Data. А также при получении параметров объекта необходимо будет осуществлять дополнительный запрос или JOIN к таблице Data, что существенно замедлит систему. Подробнее о скорости работы JOIN операций можно ознакомиться в работах [10-11].

Другой способ хранения значений параметров объекта продемонстрирован на рис. 5. На данной схеме базы данных параметры хранятся в колонке data с типом JSON, таким образом, данные объекта можно будет получать без дополнительных запросов или JOIN операций, что существенно ускорит работу механизма. Данный подход также имеет и минусы: для

извлечения данных из JSON может потребоваться дополнительная обработка на уровне приложения, особенно если данные хранятся в сложных вложенных структурах. Полнотекстовые или сложные индексы на основе JSON могут быть недоступны или работать медленно в зависимости от используемой базы данных; JSON-документы могут занимать больше места, чем структурированные данные в колонках, особенно при дублировании параметров. Способы хранения конфигураций и значений параметров объектов, описанные выше, имеют свои плюсы и минусы, и все подходят для реализации механизма управления доступами. В ходе эксплуатации механизма хранения конфигураций осуществлялось в базе данных для настройки через панель администратора (рис. 2.). А хранение значений параметров объектов было реализовано с помощью JSON колонки из-за большого количества различных параметров и типов объектов (рис. 5.). Финальная схема структуры базы данных изображена на рис. 6.

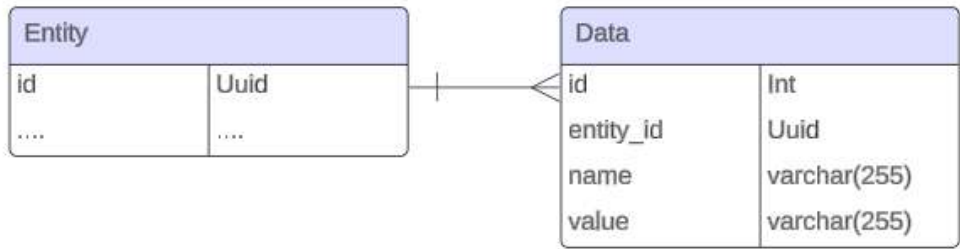


Рис. 4. Схема базы данных гибкой ролевой модели.
Fig. 4. The schema of the flexible role model database.

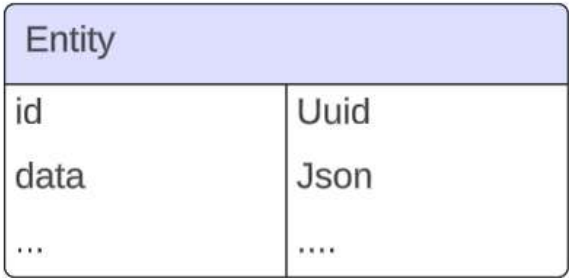


Рис. 5. Схема базы данных гибкой ролевой модели с использованием JSON поля.
Fig. 5. A flexible role model database schema using a JSON field.

Разберем назначение таблиц и полей в них. User – Реализует сущность “Пользователь”. Хранит учетные записи пользователей и их набор полей. Назначение полей:

- id – Идентификатор пользователя. Тип Uuid.
- name – Имя пользователя.
- phone – Мобильный номер пользователя.
- created_at – Дата создания пользователя.
- role – Роль пользователя. Для упрощения понимания механизма список возможных ролей сокращен до 1 роли.

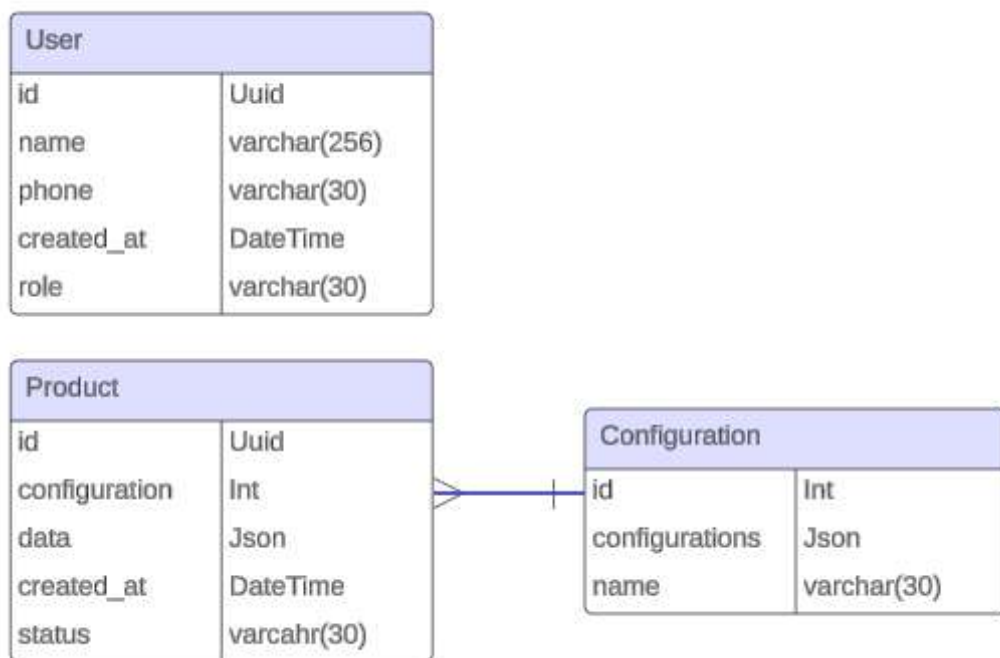


Рис. 6. Схема базы данных гибкой ролевой модели.
Fig. 6. Flexible Role Model database schema.

Product – Реализует сущность “Товар”. Хранит данные товаров, доступных на сайте. Назначение полей:

- id – Идентификатор товара. Тип Uuid.
- configuration – Тип конфигурации. Int ссылка на таблицу Configuration.
- data – Набор данных товара. JSON string.
- status – Текущий статус товара.
- created_at – Дата создания пользователя.

Configuration – Реализует сущность “Конфигурация”. Хранит данные о конфигурации сущности, требующей гибких настроек прав доступа, в данном примере будут демонстрироваться конфигурации только товаров. Назначение полей:

- id – Идентификатор конфигурации. Тип Int автоинкремент.
- configurations – Набор конфигураций. JSON string.
- name – наименование конфигурации.

Таким образом каждый товар в системе имеет ссылку на конфигурацию и набор полей в формате JSON в поле data.

Формат поля Configurations для сущности “Конфигурация” показан на рис. 7, на котором можно видеть несколько блоков. Блок data отвечает за существующий набор полей доступных для сущности, их тип и способы валидации (constraints), а также значение по умолчанию (value). Тип полей и способы их валидации могут быть как самостоятельно написанными реализациями в проекте, так и в виде распространяемых библиотек, например, Symfony Forms [12] и Symfony Constraints [13]. Блок view отвечает за права работы с полями сущности в конкретном ее статусе (STATUS), для конкретной роли пользователя (USER_ROLE) и полем (FIELD_NAME) и имеет 2 возможных разрешения view и edit. Блок permissions отвечает за список доступных действий пользователя в конкретном статусе

сущности (Status), для конкретной роли пользователя (USER_ROLE) и определённому действию (PERMISSION_NAME).

Рассмотрим несколько примеров описания конфигураций для нескольких разновидностей товаров и их статусов. Допустим в рассматриваемом интернет-магазине существует 2 вида товаров: книги и телевизоры. Книги имеют следующий набор полей: автор, цена, количество. Телевизор имеет совершенно другой набор полей: диагональ, цена, размер, количество.

Для книг конфигурация будет иметь вид, показанный на рис. 8.

```
{ "data": { "FIELD_NAME": { "type": "text",
                           "constraints": [],
                           "value": null
                         }
  },
  "view": { "STATUS": { "USER_ROLE": { "FIELD_NAME": [ "view", "edit" ]
                                     }
            }
  },
  "permissions": { "STATUS": { "USER_ROLE": [ "PERMISSION_NAME" ]
                              }
  }
}
```

Рис. 7. Формат поля Configurations для сущности “Конфигурация”.
Fig. 7. Configurations field format for “Configuration” entity.

```
{ "data": { "author": { "type": "text",
                      "constraints": [ "NotEmpty" ],
                      "value": null
                    },
  "count": { "type": "int",
            "constraints": [],
            "value": 0
          },
  "price": { "type": "int",
            "constraints": [ "min:1" ],
            "value": 1
          }
  },
  "view": { "Available": { "User": { "author": [ "view" ],
                                   "price": [ "view" ],
                                   "count": [ "view" ] },
              "Courier": { "count": [ "view", "edit" ] }
            },
  "NotAvailable": {
    "User": { "author": [ "view" ] },
    "Courier": { "count": [ "view", "edit" ] }
  }
  },
  "permissions": { "Available": { "User": [ "buy" ],
                                "Courier": [ "deliver" ]
  }
}
```

Рис. 8. Формат поля Configurations для сущности “Книга”.
Fig. 8. Configurations field format for “Book” entity.

Таким образом, тип товара книга доступен для пользователя в статусе Available для просмотра всех полей и возможности заказать, курьер не видит данных ненужных ему о книге, но может изменить доступное количество. В статусе NotAvailable пользователь видит только наименование автора и не может заказать себе товар.

Для товара телевизор отличием конфигурации будет исключительно в блоке view (см. рис. 9), то есть отличием будет исключительно набор данных, доступных для сохранения или отображения пользователю.

```
"view": { "Available": { "User": { "diagonal": [ "view" ],
                                   "price":    [ "view" ],
                                   "count":    [ "view" ]
                                   },
          "Courier": { "count": [ "view", "edit" ],
                      "size":  [ "view" ]
                    }
        },
        "NotAvailable": {
            "User": { "author": [ "view" ] },
            "Courier": { "count": [ "view", "edit" ] }
        }
    }
```

Рис. 9. Формат поля Configurations для сущности “Телевизор”.

Fig. 9. Configurations field format for “TV set” entity.

Пример реализации работы с механизмом конфигурации продемонстрирован на языке программирования высокого уровня PHP, приведенный далее код достаточно просто адаптировать для других языков программирования, например, с помощью сервиса [14]. Получение данных, доступных пользователю с помощью блока view, проиллюстрировано на рис. 10. Проверка доступности операции с сущностью с помощью блока permissions показана на рис.11.

```
<?php
/**
     $data - данные из поля data в сущности
 */
function getDataByConfiguration(array $data, array $configurations,
                                string $userRole, string $status):array
{ $returnData = [];
  $configurations = $configurations['view'][$status][$userRole] ?? [];
  // Получаем список конфигураций по статусу сущности и роли пользователя
  if (empty($configurations)) {
      return []; // В случае, если отсутствуют данные для роли и статуса
  }
  $returnData = [];
  foreach ($configurations as $key => $configuration) {
      if (in_array('view', $configuration)) {
          $returnData[$key] = $data[$key] ?? null;
      }
  }
  // если пользователю доступен view данного поля, то добавляем в ответ
  return $returnData;
}
```

Рис. 10. Описание функции получения данных блока view на языке PHP.

Fig. 10. Get data function in PHP.

Работа с гибкой ролевой моделью затрагивает не только серверную часть приложения (backend), но и клиентскую (frontend). Необходимо решить проблему отображения полей и доступности действий, например, нет необходимости отображать поле `author` для роли `Couñier` так, как оно ему недоступно. Есть несколько решений данной проблемы.

```
<?php
function isAllowed(string $rule, array $configurations,
                  string $userRole, string $status): bool
{
    $returnData = [];
    $configurations = $configurations['view'][$status][$userRole] ?? [];
    if (empty($configurations)) {
        return [];
    }
    return $configurations[$rule] ?? false;
}
// Проверяем доступна ли операция
}
```

Рис. 11. Описание функции проверки доступности операции на языке PHP.

Fig. 11. `isAllowed` function in PHP.

Очевидным решением является передача набора конфигураций на клиентскую часть приложения, таким образом, клиент или мобильное приложение в зависимости от настроек сможет генерировать форму с данными. Для этого необходимо подготовить API и передавать список доступных действий пользователя и список доступных ему полей. Плюс данного подхода в простоте реализации и единообразии обработки конфигураций на клиентской и серверной частях приложения. Минусом является передача конфигурационных настроек клиенту, чем могут воспользоваться злоумышленники для поиска уязвимостей и взлома системы.

Для исключения передачи конфигураций на клиентскую часть приложения можно применить следующую реализацию. Сервер приложения будет передавать только те поля, которые доступны пользователю для его роли. Также можно добавить новый блок настроек, например, с названием `guiActions` (рис. 12).

Зная конфигурацию, можно возвращать клиенту определённый набор действий, доступных пользователю по данному объекту, благодаря чему на клиентской стороне могут отображаться доступные пользователю блоки или кнопки.

```
"guiActions ": { "STATUS": { "USER_ROLE": [ "ACTION1", "ACTION2", ] }
}
```

Рис. 12. Пример блока настроек.

Fig. 12. Example of the settings block.

Существенным минусом механизма гибкой системы управления доступами, который предлагается автором статьи, является большой объём конфигураций и у клиента, и у сервера. В ходе эксплуатации механизма было выявлено, что получение файла конфигураций по объектам с большим количеством настроек занимало значительный объём оперативной памяти. Для минимизации потребления оперативной памяти приходилось разрабатывать дополнительные механизмы обработки и получения конфигураций, что также сказывалось на быстродействии системы.

Несомненным плюсом механизма гибкой системы управления доступами, который описан в данной статье, является возможность добавлять множество дополнительных настроек или расширений. Например, в приложении появилась потребность добавить условия перехода сущности по статусам. В качестве решения можно описать новый блок `statuses` (рис. 13).

На рис. 13 `FROM_STATUS` – это статус, с которого будет осуществляться переход, а `TO_STATUS_1` и `TO_STATUS_2` статусы, на которые возможен переход со статуса

FROM_STATUS. Наименования REASON_1 и REASON_2 – это условия перехода, это может быть как наименование класса в системе, так и код, который должен быть выполнен для проверки возможности перехода.

```
"statuses": { "FROM_STATUS": { "TO_STATUS_1": [ "REASON_1" ],  
                                "TO_STATUS_2": [ "REASON_2" ]  
          }  
}
```

*Рис. 13. Пример блока настроек.
Fig. 13. Example of the settings block.*

Разработанный механизм гибкой настройки прав доступа был успешно внедрён в нескольких проектах, связанных с управлением сложными ролевыми моделями. Его использование позволило сократить время на адаптацию системы под новые бизнес-требования, а также упростить процесс добавления новых сущностей и ролей. Например, часто в проект добавлялись новые роли, но при использовании данного механизма достаточно было обновить конфигури для работы нового типа пользователя.

Практика показала, что использование JSON-структур для хранения данных, а также чётко описанных конфигураций ролей и их прав, позволило избежать избыточности и сделать систему более производительной и масштабируемой.

4. Заключение

Предложенный механизм гибкой системы управления доступом объединяет преимущества моделей RBAC и ABAC, позволяя добиться одновременно гибкости и простоты настройки. Рассмотренные подходы к хранению конфигураций и параметров объектов в базе данных демонстрируют возможности для адаптации под различные сценарии использования, от простых интернет-магазинов до сложных корпоративных приложений.

Основным преимуществом системы является использование гибкой структуры для хранения параметров объектов, что обеспечивает скорость работы и удобство конфигурации, особенно при масштабировании. В то же время, гибкость достигается за счёт разделения настроек на права доступа к полям и действий по определенному статусу объекта, что позволяет учитывать роли пользователей, статусы объектов и конкретные операции.

Практическая реализация показала, что использование JSON для хранения данных может быть оптимальным решением для систем с большим количеством параметров и сложной логикой доступа. Однако важно учитывать потенциальные сложности, связанные с обработкой JSON на уровне базы данных и поддержанием целостности данных.

Список литературы / References

- [1]. Что такое реляционная база данных. Доступно по адресу: <https://selectel.ru/blog/relational-database>, дата обращения: 27.03.2025. / What is a relational database. Available at: <https://selectel.ru/blog/relational-database> (in Russian), accessed 27.03.2025.
- [2]. Sandhu, R. Role-based access control models. IEEE Computer Society Press, 1996, pp. 1-21.
- [3]. Ferraiolo, D. F., Sandhu, R. Assessment of Role-based Access Control Systems. IEEE Transactions on Software Engineering, 2007, pp. 1-11.
- [4]. Калимолдаев М.Н., Бияшев Р.Г., Рог О.А. Анализ методов атрибутного разграничения доступов. Прикладная дискретная математика, 2019, стр. 43-56. / Kalimoldaev M.N., Biyashev R.G., Rog O.A. Analysis of methods of attribute access differentiation. Applied discrete mathematics, 2019, pp. 43-56 (in Russian). DOI: 10.17223/20710410/44/4.
- [5]. Амелина А.А. Исследование решений для реализации ролевой модели системы контроля доступа к корпоративной информационной системе. Перспективы развития и применения современных технологий, 2021, стр. 32-38. / Amelina A.A. Research of solutions for the implementation of the role

- model of the access control system to the corporate information system. Prospects for the development and application of modern technologies, 2021, pp. 32-38 (in Russian).
- [6]. Кальдина Д.И. Ролевая модель управления доступом в системе конференций. ИТ. НАУКА. КРЕАТИВ. Материалы I международного форума: в 5-ти томах. т. 4. Молодёжь. Наука. Творчество, 2024, стр. 315-320. / Kaldina D.I. Role model of access control in the conference system. Materials of the international forum Youth. Science. Creativity: in 5 volumes, vol. 4, 2024, pp. 315-320 (in Russian).
- [7]. Мухаммад А., Миков Д.А. Ролевая модель управления доступом в системе динамического архивирования. Инновационные научные исследования, 2021, с. 109-203. / Muhammad A., Mikov D.A. A role-based access control model in a dynamic archiving system. Innovative Scientific Research, 2021, pp. 109-203 (in Russian).
- [8]. Иргашева Д.Я., Усманов А.К. Разработка ролевой модели с зональным разграничением доступа. Наука и мир, 2016, стр. 35-40. / Irgasheva D.Ya., Usmanov A.K. Development of a role model with zonal access control. Nauka i Mir, 2016, pp. 35-40 (in Russian).
- [9]. Документация инструмента AWS S3. Доступно по адресу: <https://aws.amazon.com/ru/s3>. Дата обращения: 27.03.2025. / Documentation of the AWS S3 tool. Available at: <https://aws.amazon.com/ru/s3>, accessed: 27.03.2025.
- [10]. Исследуем производительность JOIN в MySQL. Доступно по адресу: <https://habr.com/ru/articles/122210/>. Дата обращения: 30.11.2024. / Let's explore the performance of JOIN in MySQL. Available at: <https://habr.com/ru/articles/122210/>, accessed: 30.11.2024.
- [11]. Как реляционная СУБД делает JOIN? Доступно по адресу: <https://habr.com/ru/articles/560834/>. Дата обращения: 30.11.2024. / How does a relational database do a JOIN? Available at: <https://habr.com/ru/articles/560834/>, accessed: 30.11.2024.
- [12]. Веб-сайт Symfony Forms. Доступно по адресу: <https://symfony.com/doc/current/forms.html>. Дата обращения: 06.11.2024. / Symfony Forms website. Available at: <https://symfony.com/doc/current/forms.html>, accessed: 06.11.2024.
- [13]. Веб-сайт Symfony Constraints. Доступно по адресу: <https://symfony.com/doc/current/reference/constraints.html>. Дата обращения: 06.11.2024. / Symfony Constraints website. Available at: <https://symfony.com/doc/current/reference/constraints.html>, accessed: 06.11.2024.
- [14]. Онлайн конвертер кода. Доступно по адресу: <https://syntha.ai/>. Дата обращения: 04.11.2024. / Online code converter. Available at: <https://syntha.ai/>, accessed: 04.11.2024.

Информация об авторах / Information about authors

Тимур Игоревич ВАСИЛЬЕВ – выпускник Казанского национального исследовательского технического университета – КАИ имени Туполева в городе Казани, где получил степень магистра по специальности "Информатика и вычислительная техника". В настоящее время является ведущим разработчиком в финтех компании Paybis.com.

Timur Iгореvich VASILIEV got his Master degree in Computer Science and Engineering from the Tupolev Kazan State Technical University in Russia. Currently, he is a leading developer in a fintech company Paybis.com.

DOI: 10.15514/ISPRAS-2025-37(4)-9



Managing Master Data Implementations

S.V. Kuznetsov, ORCID: 0000-0001-6752-6742 <sergey.kouznetsov@gmail.com>

D.V. Koznov, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

D. V. Luciv, ORCID: 0000-0002-6332-2360 <d.lutsiv@gmail.com>

*Saint Petersburg State University
7/9 Universitetskaya emb., St. Petersburg, 199034, Russia.*

Abstract. Every business organization has a subset of data which must be highly consistent: legal information, supplier and contractual data, customer base, etc. Customers and employees expect to receive the same information about the same data object from different organization sources, which are usually other information systems. The process of consolidation and centralized control of such data throughout the organization is called Master Data Management (MDM). The iterative deployment strategy is a popular way to introduce MDM to a organization that supposes a step-by-step implementation of MDM components based on the real needs of the organization. In this paper, we present a functional MDM model for the early stages of MDM implementation within the iterative deployment strategy. The purpose of this model is to represent real business needs of an organization in terms of MDM, making clear which MDM components should be implemented, and which should not. Detailed description of the model components is provided. Also, a case study, presenting a portfolio of six real MDM projects analyzed from the viewpoint of the proposed model is performed.

Keywords: enterprise applications; master data management; open source; component approach.

For citation: Kuznetsov S.V., Koznov D.V., Luciv D.V. Managing Master Data Implementations. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 1, 2025. pp. 161-176. DOI: 10.15514/ISPRAS-2025-37(4)-9.

Управление MDM-проектами

С.В. Кузнецов, ORCID: 0000-0001-6752-6742 <sergey.kouznetsov@gmail.com>

Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

Д.В. Луцив, ORCID: 0000-0002-6332-2360 <d.lutsiv@gmail.com>

*Санкт-Петербургский государственный университет,
Россия, 199034, Санкт-Петербург, Университетская наб., д.7-9.*

Аннотация. Управление мастер-данными (Master-Data Management, MDM) – особый вид управления данными бизнес-организации, нацеленный на идентификацию, очистку, консолидацию и централизованное управление важнейшими данными компаний (golden dataset), которые обычно распределены по разным информационным системам и другим источникам. Ведущие мировые поставщики программных решений (IBM, Oracle, Informatica и многие другие) предлагают широкий спектр готовых продуктов по управлению мастер-данными (MDM-продукты). Однако внедрение MDM сопряжено с большими затратами: необходимо не только адаптировать эти продукты под особенности организаций, но и выполнить изменение бизнес-процессов, создать новые политики работы с данными, решить вопросы безопасности, решить другие вопросы. В связи с этим в России популярна итеративная стратегия внедрения MDM, подразумевающая поэтапную реализацию управления мастер-данными на основе реальных нужд организации-заказчика. В работе вводится понятие MDM-решения, которое является результатом внедрения MDM в организацию и включает помимо программной части также обработанные мастер-данные, налаженные организационные процедуры по сопровождению мастер-данных и прочее. Основным результатом статьи является функциональная модель управления мастер-данными, предназначенная для ранних стадий разработки MDM-решения в рамках итеративной стратегии. Целью данной модели является отобразить реальные потребности организации на язык MDM – важно понять, какие именно MDM-компоненты должны быть реализованы в первую очередь. В работе также приводится экспериментальное исследование, в котором уже реализованные MDM-решения анализируются с помощью предложенной модели.

Ключевые слова: корпоративные информационные системы; мастер-данные; открытые системы; компонентные системы.

Для цитирования: Кузнецов С.В., Кознов Д.В., Луцив Д.В. Управление MDM-проектами. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 161–176 (на английском языке). DOI: 10.15514/ISPRAS-2025-37(4)–9

1. Introduction

Converting to digital economy requires business data management in to be as universal as, for example, accounting [1]. It is impossible to imagine a normally functioning organization that does not properly support special data such as legal information, supplier and contractual data, a customer base, etc. Its users rely on its consistency within the organization, i.e., they expect to receive the same information about the same object from different sources, which are mainly the information systems within the organization. Inconsistencies and contradictions in the data lead to problems, delays, and collisions, as well as financial and public image losses [2]. This special data, when consolidated and properly maintained, is commonly called Master Data¹, and, consequently, the process of its consolidation and maintenance is called Master Data Management (MDM) [1, 8].

¹ Reference data, such as registers, classifications, directories, etc., is often distinguished as a special type of master data. In Russia, handling of reference data is highly developed: there is a large volume of such data in large organizations, as well as a wide set of standards (federal, industrial, and regional) and solutions [3–7].

MDM includes a wide set of methods and strategies which have been captured and listed in a structured fashion within the DAMA-DMBOK (Data Management Body of Knowledge) [2]. There is also a large number of ready-made MDM products by well-known vendors such as IBM, Oracle, Informatica, etc. (see, for example, Gartner's Magic Quadrant report 2021 [9]). However, MDM implementation is currently not standardized and well-elaborated, since its target organizations, especially large ones, possess a lot of various individual features.

In practice, there are two fundamentally different strategies of MDM implementation: top-down and iterative [2]. The top-down strategy supposes the following chain of actions: creating a strategic conceptual MDM framework for the organization, specifying the MDM requirements, customization and fine-tuning already existing MDM products, performing necessary administrative work, and finally, running and maintaining the MDM solution. The iterative strategy, in its turn, supposes that MDM is implemented to solve a specific important problem of the organization, assuming the following extension of the MDM functionality and/or implementation of MDM for other businesses of the organization, i.e., to solve other business cases. These strategies correspond to the types of the organizational innovation proposed in [10]. The first one can be compared to technology push: innovation consists of the integrating cutting-edge technology intended to solve various, including even currently unknown, problems within the organization. The second one is comparable to the organization pull: the organization, or more precisely, its specific needs, initiate the innovation. Not rejecting the first strategy, we focus on the second one. It is less risky and allows to achieve specific practical results within manageable timeframe.

During iterative deployment of an MDM project, the organization's requirements need to be translated into the MDM terminology, which is highly advanced and has shown considerable formalization progress [2]. If these needs of the organization translate well into MDM, then it can be solved using existing MDM tools [9], which significantly cuts the costs of the project. However, it is often the case that the organization is not well-versed into MDM terminology, and tries to propose other types of projects or request an implementation of an MDM project from the scratch. Such mistakes lead to collisions, extended deadlines, and financial loss.

In this article, we propose a functional model of master data management that is intended to support the process of preliminary estimation and coordination of the work in the iterative MDM projects at their earliest stages. Note that we are primarily focused on creation/deployment of the IT infrastructure that supports MDM, as well as setting up and performing necessary analytical work (data cleaning and consolidation, classification and hierarchization, etc.). Further on, the organization will perform this analytical work on a regular basis using the deployed infrastructure, and its complete automation should be the ultimate target. Focusing on the software and analytical aspects of MDM, we purposefully omit important aspects, such as the modification of the business processes and data handling practices, as well as employee training and other issues. In our experience, they are solved much better when MDM software is fully deployed and implemented. Furthermore, there is a discipline dedicated to these topics – Data Governance [11]. Below we use the proposed model to describe several industrial MDM projects that we have participated in, demonstrating its applicability in practice.

The approach was briefly outlined in [12]. In this paper, we explain our ideas in more detail, including adding new examples. Moreover, the experimental study was extended with additional attributes of MDM projects, namely, overall data volume in a company, master data volume, number of data sources, project work distribution (software development, analytical work, and data processing).

The rest of the paper is organized as follows. Section 2 presents a brief overview of the master data concept. Section 3 provides the definition of the master data management projects. We introduce the MDM solution in Section 4 as a result of an MDM project, and describe its various compounds. The proposed Functional Master Data Management Model is described in Section 5. In Section 6, we present a case study with completed real-life industrial MDM projects, in which the proposed

model was employed. Section 7 contains an overview of the related work. Finally, we present our conclusions in Section 8.

2. Master Data

Following the Gartner Glossary [8], master data is a special type of business data that describes most essential characteristics of an organization, such as its current and potential customers, suppliers, consumed and manufactured products, office and production sites, billing information and details concerning accounts of individuals or counter-parties that it deals with, etc. The point of distinguishing master data as a separate concept is that medium and large organizations possess variety of information systems and use many external heterogeneous data sources. In the result, the same information ends up being represented by different data and in different formats. Furthermore, various sources may contain different attributes for the same data, and the data itself may be contradictory. In general, this can be tolerated to certain extent, but there is a subset of data which cannot tolerate disorganization at all: critical data, inconsistencies in which harm the organization and inhibit its normal functioning. This particular type of data receives special treatment thus becoming master data.

Therefore, an organization should launch a dedicated process of *Master Data Management (MDM)* for efficient consolidation, usage and maintenance of the master data. MDM supposes that a organization properly cares about its business processes, quality control and integration of data, as well as standardization of the existing information systems [13]. In other terms, it focuses on collecting and accumulating data from various sources, i.e., information systems that exist within the organization (data sources, DS), additional consolidation of this data and its distribution (delivery) to the consumer information systems (data consumers, DC).

DAMA methodology [2] identifies the following key steps of an established MDM process within an organization: (i) data model management, (ii) data collection and accumulation, (iii) data validation, standardization, and enrichment, (iv) entity and data inconsistency resolution.

MDM should be supported by a special IT solution created and implemented within an organization (further referred to as an *MDM solution*). An integral part of an MDM solution is a central repository or *data hub*. It collects master data candidates which are then appropriately processed and delivered to DC systems. Gartner identifies four approaches to the Data Hub architecture [9]:

- *Registry*. The hub does not contain the data itself, but only the corresponding references (indices). This approach is relevant for the data that cannot be copied or "moved" for various reasons, e.g. data under certain regulation.
- *Consolidation*. Data is uploaded into the common repository on a regular basis, appropriately processed, and then the hub itself provides DC systems with an access to this data. Here new data is uploaded into the hub on a regular basis by DS.
- *Centralization*. This architecture is very similar to the previous one, but the hub takes over data input as well: i.e., data could be input directly to the hub itself, and thus turning all systems that initially were DS into DC.
- *Coexistence*. This architecture implements a combination of the Consolidation and Centralization for different master data of an organization, i.e. some information systems could play both roles of DS and DC. Additionally, if some data fragments are not "movable", they can be handled using the Registry approach.

A wide set of ready-made software tools to create MDM solutions already exists. First of all, there are so-called "boxed" products such as SAP MDG, Informatica MDM, IBM InfoSphere MDM and others, which are focused on solving standard tasks of master data management. However, these tasks can vary to a large extent. This has motivated some vendors, including some of the abovementioned ones, to offer "software construction kits" which can be pieced together for a given need, for example, Informatica MDM, Unidata, and others. Thus, implementing an MDM solution for a specific organization remains a complicated and labor-consuming effort.

3. MDM Projects

In practice, MDM projects often start as common IT projects. Therefore, it is necessary to identify whether the given needs of an organization are clearly MDM-oriented, which allows using ready-made MDM products.

An organization clearly needs MDM when it finds necessary to perform data collection, enrichment and consolidation from various DS systems, as well as the delivery of this data to various DC systems, which then use it to support various business functions. DS can be both internal and external: for example, an organization may need to enrich its customer data with information collected from social media. The requirement for several DCs receiving master data is less strict and sometimes may be omitted. However, in this case the organization runs a critical business process that requires high-quality enriched consolidated data obtained from various DS systems. For example, processes such as validation of a new client or of a suspicious transaction in a bank.

Let us also consider when an organization does not need an MDM project. Firstly, when the processed data is homogeneous: for example, it was entered into an information system or set of systems manually by operators (operator input). These projects may require a logical data model, data cleaning and validation, various modes of data access, and so forth. But they lack the main MDM task of the data consolidation from different sources. Secondly, when complicated business logic is required: this type of functionality should be moved out of the MDM solution into separate information systems [9]. An MDM project ends at the delivery of the master data to DCs.

In summary, MDM-specific tasks must be the central focus of the project. If it is not the case then this particular IT project is not an MDM one.

4. MDM Solution

As a result of an MDM project, the organization receives an *MDM solution*. It includes the following: (i) implemented MDM-product as *MDM-system* (the software), (ii) new data handling rules, (iii) trained employees, and (iv) an established and running Master Data Management process. The last point is of the same importance as the other ones: the business process may not start following MDM process due to other problems, e.g. security issues, high workloads of employees that were assigned to with various MDM-specific tasks in the process, etc.

Let us describe in detail what an MDM system is. First of all, it is deployed software system that implements the main MDM functionality: data hub, consolidation and survivorship rules, etc. The main part of this software system is the customized MDM product (see the list of the available products in [9]). Additionally, it may include a set of utilities and various tools that perform particular tasks (such as data cleansing). Availability of the ready-made multifunctional software, which requires just configuration and deployment within the organization significantly decreases the costs and risks of the project. However, some compounds of an MDM system have to be implemented separately within the MDM project in order to represent the specifics of the organization that could not be covered via standard tools. MDM is commonly implemented in large and complex organizations that have already established their own processes and obtained their own unique traits. Therefore, a completely ready-made software solution that is suitable for any given task or business case does not exist and MDM products need to be modified to meet all organization needs.

5. Model Description

It is necessary to perform more detailed analysis is required after a preliminary informal discussion of the organization's needs in order to find out how to precisely express them via MDM [8, 9]. We propose a special functional model to increase the efficiency of this process. It describes a typical MDM solution, including an all-encompassing functionality set, so that its users are able to choose which functions they need to implement to serve their particular needs. The proposed model can be

called an ideal model to-be: this term is common in structured [14] and object-oriented analysis [15], as well as in business process re-engineering [16].

For the ease of use, the model is represented via a full lifecycle metaphor of master data. In software engineering research, a metaphor is an analogy drawn with physical reality in order to elaborate on virtual abstractions. In this case, we will use a model life cycle of master data as a metaphor, keeping in mind that a real life cycle is a much more complicated process. However, the package-steps that we have identified allow to conveniently structure the functional components of a typical MDM solution and use the model for planning the functionality of the specific MDM solutions.

Our model consists of three packages (steps): data collection, data processing, and data delivery. Packages contain functional components each describes a set of activities concerning setup or management of the master data. Therefore, components of our model include both MDM implementation and its further support, i.e. the activities performed during the following functioning of the MDM solution.

For example, in order to implement Data Consolidation, the following activities are required:

- **MDM implementation:** customize/configure existing software that supports the analyst's workplace, define the rules for conflict resolution, and consolidate the data from organization DS systems for the first time.
- **MDM support:** further data consolidation as one of the MDM solution functions, since the data from DS systems will continue to flow to the data hub.

Discussing MDM projects, we strictly focus on *software development* and *analysis* aspects of MDM. The software part of an MDM project includes customization of a ready-made MDM product, which can be configured and modified for the organization's specifics, and development of the special software for some particular features such as access to data of existing information systems in an organization, data quality scripts (e.g. special cleaning rules which are applicable for this particular data of this particular organization), software to resolve consolidation conflicts (e.g. introducing appropriate machine learning algorithms), etc. Analysis aspects refer to the establishment and initial performance of the analytical work: data cleansing and consolidation, classification and hierarchization, etc.

It should be noted that we omit other activities such as business processes modifications, establishment of new data policies, employee training, etc. On one hand, the latter are usually performed by the organization itself, as the MDM project is implemented within the iterative strategy, and therefore is an answer to the organization's specific request. Thus, the organization should be able to support the implementation by providing these required additional activities. On the other hand, administrative MDM issues can lead to the further questions of the data management, which are covered by Data Governance [11], another well-known domain.

Thus, every component of the model has a *software* part and an *analytical* part. For example, creating master data logical model is analytical work only, but data cleansing concerns both parts. The latter could be stated about implementing special cleansing rules, and their usage, including result analysis, leading to potential other rules, etc. Usually, MDM products and some ready-made tools are used but some steps specific to an organization can be implemented as separate scripts, this is so-called additional software created withing the MDM project.

Let us now describe the main packages of the model.

- **Data Collection.** This package includes components that identify master data candidates (raw data), and perform its further analysis and preprocessing. Furthermore, it also includes the implementation of the DS systems access.
- **Data Processing.** This package contains functionality to provide development and storage of the master data in the data hub, including creation and maintenance of a logical data model, as well as classification, hierarchy building, and so on. Therefore, the data hub receives raw data from DS that has been preprocessed by the previous package. This package processes it, transforming into the master data.

- **Data Delivery.** This package contains components that deliver master data to DC systems. Note that DC and DS systems can coincide, fully or partially. This emphasizes the issues of the distribution master data access rights and various data delivery modes. We have identified the following modes of the master data delivery: (i) package-driven, (ii) real-time, (iii) subscription-based.

It is important to note that the proposed model is focused on incremental/repeating master data updates in the data hub that take into account new raw data, instead of a single-time upload.

Packages and their functional components are presented in Fig.1.

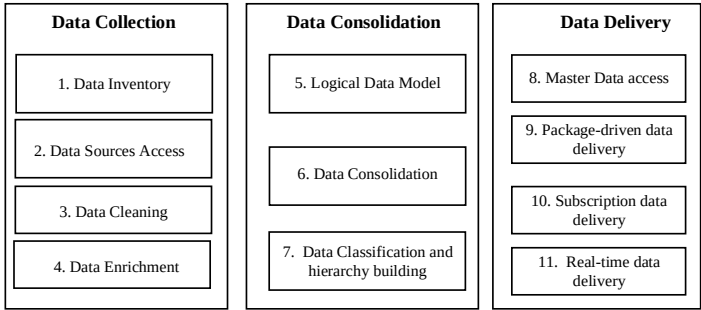


Fig. 1. A schematic representation of a typical MDM solution.

5.1 Data Inventory

This component identifies DS systems and determines which organization's data in particular is to become master data. It is important to define precisely what master data will consist of because as the variance in data increases, the complexity and costs of the MDM project increase as well. Furthermore, only those attributes that the DC systems are demanding should be collected at this stage. Accessing some data may be challenging due to its security policies, or organizational and technical access complications.

It is essential to determine data types, real values range of each attribute, their default values, etc. This package also identifies the trust level for each DS. Some DS systems may have a very low trust level, for example if their data have not been updated for a long time. Therefore, they should be accessed only in extreme cases.

This functional component is predominantly analytical.

5.2 Data Source Access

Since the raw data used to create master data resides in the DS systems of the organization, an MDM solution requires an implementation of the software-based access to this data in order to, at least, upload it into the hub. In most cases, data upload is a repeating procedure that is performed regularly by the functioning MDM solution. In order to automate it, considerable volume of technical work is required: DS systems may be outdated and may not even have external software access interfaces.

This functional component is predominantly software-based. The scope of work largely depends on the level to which internal data exchange has been established and developed within the organization. The organization may have already implemented data "transportation" layer between various information systems, for example, based on an enterprise data bus.

5.3 Data Cleaning

This component concerns error detection and potential fixing as well as normalization of the input data from different DS systems before its upload to the data hub. This step is important since duplicate detection and data consolidation would become quite challenging without it. Data cleaning is a high effort process, and 100% result is rarely achievable. In any case, it is reasonable to perform

preliminary cleaning that includes normalization, format unification of the significant attributes. This type of cleaning is not absolutely required, but it drastically simplifies further essential steps of data consolidation and entity resolution.

This component is both software and analytical. Despite of a large body of ready-made software it is often the case that either a tool needs modifications to correctly process various data formats in question, or custom data cleaning logic needs to be implemented. For example, a DS system may store several values in a single attribute, and these values have to be split into the corresponding fields, which requires specialized software utility/script development.

5.4 Data Enrichment

An organization may need to augment its master data contained in their DS systems with some open-source data. A prime example is supplementing contractual data with legal and tax-related information. This component is software and analytical since it requires analysis and annotation of the corresponding data obtained from DS, as well as implementation of the software-based access to the public sources used for data enrichment.

5.5 Logical Data Model

This component is intended for development and maintenance of the logical model of master data. The model needs to represent the structure of the consolidated data containing all the attributes collected from the DS systems of the organization. It is essential for further processing of the master data, as well as its delivery to DS systems. One of the most important steps in the logical data model creation is the reconstruction/detection of various relationships within the data that are absent in the DS systems, but to be appeared during the consolidation process.

Logical data model creation, which is identifying which entities, attributes, and relationships should appear in the new schema, is an analytical process. However, it should be supported with modelling software and visualization tools that operate on a list of attributes and entity relationships (often of different types), as well as a software link of the master data model to the corresponding mapping of its elements to DS and/or DC systems. At the same time, some aspects of these tools may need to be modified for the given project.

5.6 Data Consolidation

This component is responsible for uploading data from DS systems to the hub and consolidating it according to the previously created logical model. This process is performed automatically; however, data collisions or conflicts arise during its execution, which can be resolved in the following ways.

- "Manual" approach involves a domain or subject area expert to resolve the data conflict. It is employed when automatic conflict resolution is inappropriate: i.e., for critical data (such as legal) that cannot tolerate errors.
- Machine learning techniques, which learn typical situations in order to automatically resolve consolidation conflicts.
- Combined strategy: domain experts can definitively resolve preliminary results provided by machine learning algorithms. This approach can drastically lower the conflict resolution efforts.

Raw data can be uploaded to the hub from DS systems once, for example, in case the Centralized architecture is used, or, in case there are DS systems that discontinued their function but still contain valuable data. Otherwise, besides the initial raw data load, incremental uploading procedure should be implemented.

This component is both software and analytical. Its software part is the analyst's data consolidation workplace, which often has to be modified to support handling specialized data and implementation of custom data consolidation and conflict resolution rules that could be used automatically.

Additionally, if machine learning algorithms are used then they should be trained correspondingly, e.g. they could be used to implement updatable or self-configuring rules of data conflict resolution.

5.7 Classification and Hierarchy Building

Organizations require their master data to be as systematized and ordered as possible. Often it is necessary to perform data *classification* that means defining data clusters, and identifying their significant characteristics. At the same time, organization's data may need to be linked to the external classifications such as federal standards, industrial classifications and so on. In many cases data also requires *hierarchization*: for example, reconstruction of the hierarchy of customers or suppliers. Consider a case when a new customer of the organization belongs to the same branch of a large corporation as another one, and thus there is no need to perform new security clearance, which saves a lot of time and resources during contract preparation. Another example of the data relationship: if it is known that two customers are married, the organization can offer them additional services. Classification and hierarchy building is performed on the unstructured data, and thus it can be combined with data enrichment.

This component is analytical concerning the definition of the rules of hierarchy building and classification. It should be supported by the appropriate software that allows to "test" these rules on a small volume of data first, and only then applies them to the whole set of data. At the same time, various machine learning methods are increasingly employed as analytical recommenders and semi-automatic data classifiers. Almost everything in the toolset of this package requires modifications and configuration due to business cases and data specifics.

5.8 Access Rights Model

Different data consumers can be situated in various business segments of the organization and therefore have different data access rights. For example, some consumers should be able to access all data, but others should not see certain attributes. This component defines and implements a strategy for master data access rights. It should be based on the existing user roles within the organization and corresponding data access rights, and it requires interacting with the security team.

This component includes tasks that are mostly analytical and not time-consuming. The policies of master data access are implemented via the means of the information system security management. However, creating a corresponding specification is a highly significant task that requires deep knowledge of the data and business processes, as well as the structure of the organization. Data security granularity and its efficient implementation are the key to the success of the MDM project.

5.9 Package-Driven Data Delivery

This component is responsible for uploading and updating master data in the DC systems according to a given schedule. Many DS systems focus on uploading master data into datamarts, which they are further handling according to their own procedures. In general, each datamart uses its own master data fragment. It is reasonable to implement a separate management tool for datamarts in order to refresh its master data, monitor and audit the DC queries. This approach allows to track not only what master data is consumed by which DC but also what conflicts arise and how they correspond to the master data consumption.

This component is mainly software including implementation and configuration of an MDM system's interface to send data to the appropriate DSs or utilize corresponding ETL procedure(s)². The analytical part is not too complicated and consists of determining which DCs and master data fragments that require this particular strategy.

² ETL (Extract, Transform, Load) is the general procedure of extracting data from source information systems, transformation of the extracted data and its delivery to the destination systems [17]. MDM may be one of the boxes of the ETL procedure.

5.10 Subscription-Based Data Delivery

Within this mode, every DC system subscribes to a particular fragment of the master data (fragment of the logical model or a subset of entities and their attributes). Then one or several queues receive the newest version of this master data after each update. Finally, all the DC systems load their updates from the queues according to their subscriptions. This model's convenience lies in the hub being a centralized place of management and administration of the master data across all DCs. The complexity of its implementation is that either the existing queue mechanism used by DCs has to be reused, or the DCs have to be modified in order to use MDM solution queues. This component is software and analytical at the same time similar to the previous one.

5.11 Real-Time Data Delivery

This component consists of master data delivery to DCs in real time, i.e., immediately after the master data has been modified (consolidated and updated). This mode is often hard to implement due to conflicts that arise on the consumer's side as the DC system can temporarily block access to the data fragment during some operations execution, which leads to delays in the master data updates in the hub. This component is both software and analytical.

6. Case Study

In the presented case study, we consider organizations that belong to different business sectors (the energy industry, telecommunications, transportation, retail, government, and machine manufacturing industries) in order to demonstrate that the iterative approach and the proposed functional model are universal and independent of the subject area in question. The list of the organizations and their characteristics is presented below. We have considered medium-size and large business organizations (i.e. more than 10 000 employees) because the iterative approach to MDM is most relevant for them. Due to their large size, these organizations contain many various data domain segments which are often located in the separate departments. In their turn, the latter may be independent subdivisions of the whole organization having their own local information systems and data. The iterative approach enables MDM implementation not for the whole organization but for selected individual department considering limited data domain segments and based on the department business needs and scenarios. After successful completion of the first MDM project the following MDM project may be initiated in the organization for another department. Departments and their domain segments considered in this case study are presented in Table 1.

The original goals and business case(s) of the studied MDM projects (let's code the MDM solutions of these projects in bold to reference further in this work) are described below.

- **IP**: Producing a global inventory catalog in a large corporation in the energy sector (sales department).
- **PIM**: Consolidation of technical and marketing information to produce a product catalog within a telecommunication organization (sales department).
- **PSC**: Creation of catalogs of products/services and customers/suppliers for procurement purposes, management of the relationships between customers and suppliers (supplier department).
- **CS**: Enrichment of a clothing retailer's customer database with social media data in order to identify and support influencers (customer support department).
- **CPO**: Personal data consolidation across regional and federal databases – real estate, vehicles, etc. (citizen communication department).
- **CDV**: Identification and verification of the customer data for a global corporation; management of legal and related hierarchies. Support of the customer's segmentation (customer onboarding department).

To meet these requirements the MDM projects were conducted within the corresponding department of the organizations, which characteristics are described in Table 2.

Table 1. Studied Organizations.

Organization code name	C1	C2	C3	C4	C5	C6
Business sector	Energy	Telecom	Transport	Retail	Government	Energy & machine manufacturing
Organization size (number of employees)	48 800	36 000	729 000	14 630	100 000	184 000
Organization department involved	Sales	Sales	Supplier	Customer support	Citizen communication	Customer onboarding
MDM domain segments selected	Inventory	Products, personal data	Products, services	Products, personal data	Personal data	Customer data

Table 2. Real-life MDM projects.

№	Project code	Org code	Overall data volume (K records)	Master data volume (K records)	Number of data sources	Project sizing (man-months)	Project duration (months)	Project work distribution, %		
								sw	analysis	dp
1.	IP	C1	500	350	15	9450	18	23	70	7
2.	PIM	C2	100 000	70 000	12	3024	12	30	30	40
3.	PSC	C3	2 000	700	4	7560	24	30	45	25
4.	CS	C4	70	30	5	1890	9	16	64	20
5.	CPO	C5	170 000	30 000	15	4536	12	45	25	30
6.	CDV	C6	4 000	1 500	213	10080	24	56	24	20

- “Overall data volume” indicates the overall data volume in these segments measured in thousands of records.
- “Master data volume” presents the resulting master data volume, extracted and consolidated in the context of the studied MDM projects. It is obvious that not all of the domain segments’ data should be included in the master dataset, i.e. be considered as master data. The volume of these data is also measured in thousands of records.
- “Number of data sources” describes the number of data sources for every MDM project. This is one of the primary characteristics for determining whether the MDM approach is suitable to address the needs of an organization as MDM is focused on consolidating data from various sources. If, however, there is a small number of sources, consolidation is unnecessary and therefore, other means should be used to satisfy the organization needs.
- “Project sizing” indicates the overall estimation of the MDM projects (in man-months).
- “Project duration” shows the calendar duration of these projects (in months).
- “Project work distribution” shows percentage distribution of the total work between software development (sw), analytical work (analysis), and actual data processing (dp) including all the uploads, incremental loads, etc.

Note, that MDM projects are actually smaller than projects of information system development from the scratch. Moreover, as we can see in, the ratio of software development in these projects is 33% on average, analytical work is the most significant – 43% on average.

It should be noted that the authors of the article have actively participated in these projects, utilizing the proposed functional model for defining the key aspects of MDM projects. Table 3 shows which components of the model were implemented in the specified MDM projects. The following scale was used:

- *High* denotes that the component is crucial for the project (business-critical or technologically complex)
- *Med(ium)* refers to a component that is necessary, but was not high-priority or effort-demanding
- *Low* denotes a component that has a "light" version: it either existed within the organization previously and required only modifications within the given MDM project, or was moved out into a separate project
- *N/A* means that the component was not required for the project

Let us consider the data presented in Table 3 from the MDM project's perspective.

- **IP:** One of the project's business requirements was to automate complex organization data handling policies that were employed in more than ten various departments. Implementing a material resource classifier (*data classification and hierarchy building*) as well as creating a *logical data model* were its key points.
- **PIM:** The project was mainly focused on the *data inventory* of the organization products taken from various DSs, creating a unified *logical master data model*, and *data consolidation*. Besides, it required the construction of a product tree that contained all product information, including financials, so the sales department and financial experts would be able to perform further analysis (*data classification and hierarchy building*).
- **PSC:** The project addressed *consolidation of data* of the goods purchased by the organization. It was necessary to combine all the information about purchased items obtained from various commodity nomenclatures, and to create a list of contractor services. Besides data consolidation, the project was also focused on the *access rights model* and *subscription-based data delivery*.
- **CS:** The project required *data enrichment* and *consolidation* with the purpose of identifying top influencers on social media among the customers of the clothing retail organization to provide them with some special treatment and preferences.
- **CPO:** This project was developed for a government management service in order to create a "smart" personal account of a citizen. It was necessary to integrate the account within federal and regional information systems. The particular goals of the project were information security and corresponding *access rights model*, as well as *real-time* and *subscription-based data delivery*.
- **CDV:** The organization her had hundreds of thousands of customers worldwide, and so its customer onboarding procedure was very complicated. Before the MDM project got implemented, the procedure had taken 21 days, but after it got moved to production, the duration decreased to just 8 days. This project was focused on *data inventory* and *logical master data model*, which facilitated duplicate detection of legal entities and the search of their affiliated entities. Besides, it's required *real-time master data delivery* in order to speed up the target business process.

So, we can conclude that the proposed MDM functional model objectively describes organization's needs and allows to consider MDM projects significantly decreasing the costs and risks of their implementation. Additionally, the proposed model appears to be convenient to identify high level functional components of these projects. Besides, it can be seen that different MDM projects have significantly different focal points, and many model components are marked as "low" or "N/A", i.e., they were very easy or omitted altogether, which also has cut implementation costs.

Table 3. MDM Solution functionality.

Functional Model Components	Target MDM-solutions					
	IP	PIM	PSC	CS	CPO	CDV
I. Data Collection						
Data Inventory	Med	High	Med	N/A	Med	High
Data Source Access	Low	Med	Low	Low	Low	Med
Data Cleaning	Med	Low	Med	Med	Med	Med
Data Enrichment	N/A	Med	Med	High	N/A	N/A
II. Data Processing						
Logical Data Model	High	High	Med	Low	Med	High
Data Consolidation	Med	High	High	High	Med	Med
Data Classification and Hierarchy Building	High	High	Med	N/A	N/A	Med
III. Data Delivery						
Access Rights Model	Low	Low	High	Low	High	Med
Subscription-Based Data Delivery	Med	Low	High	Low	High	Med
Real-Time Data Delivery	Med	N/A	Med	Med	High	High
Package-Driven Data Delivery	Med	Med	Med	N/A	Low	Med

7.Related Work

In recent years, a significant number of standards and data management methodologies has emerged: DAMA-DMBOK [3], CMMI Data Management Maturity Model (DMM) [18], IBM Data Governance Council Maturity Model [19]. MDM is included in these standards, and DAMA-DMBOK considers it most extensively.

The following are the DAMA-DMBOK models that concern master data management.

- "Context Diagram: Reference and Master Data". This model defines the intents of data management, describes all the necessary activities (see the following model), their inputs and outputs, identifies suppliers, participants, and consumers, introduces the notion of a business driver and a technical driver, and finally, defines the necessary methods, tools, and metrics.
- Model that describes activities for MDM implementation: (i) identifying drivers and requirements of MDM, (ii) analyzing and evaluating data sources, (iii) selecting data hub architecture, (iv) modeling master data, (v) implementing master data management and integration; (vi) defining control policies and ensuring compliance.

"Context Diagram: Reference and Master Data" is a comprehensive vision of the MDM framework. Nevertheless, it is predominantly a descriptive model that provides a conceptual framework rather than a tool suitable for a practical use.

Let us consider in detail the model of MDM implementation activities within an organization. Despite its rather pragmatic viewpoint, it omits the issues concerning software. In general, the vision behind this model is based on analysis and administrative work, while we substantially highlight the role that software plays in MDM implementation.

Another type of existing studies is focused on MDM software in particular. Generally speaking, these studies consider MDM product adoption methodologies which every MDM manufacturer possesses: a good example would be the Velocity Methodology by Informatica [20]. These methodologies are focused on adoption of specific products of the selected manufacturer to the organizations. In contrast, the model proposed in the current paper is independent from any particular MDM product.

Among the numerous papers by Gartner who widely analyze questions of MDM implementation, [21] can be highlighted. It proposes a framework consisting of seven components: vision, strategy, metrics, information governance, organization and roles, information life cycle, and enabling

infrastructure. Similarly to our model, it is intended for the early stages of MDM implementation. However, the scope of this model is the conceptual MDM implementation within an organization, i.e., it assumes a top-down strategy. In contrast, our model is intended for the iterative strategy focused on satisfying specific organization needs.

An example of an academic MDM research is presented in [22], which proposes a model for analysis of the master data life cycle in a organization. It is intended for analyzing the existing master data management cycle, i.e., identifying missing activities and bringing to light its challenges. The main components of this model are: (i) data portfolio (organization data that undergoes master data management); (ii) data and system design (data hub architecture, IT-related aspects of MDM) (iii) data management (development and maintenance of the master data (including its updates), quality control); (iv) data maintenance (monitoring and controlling the MDM process). Note that the proposed model does not consider the software aspects of MDM solutions. Furthermore, it is not focused on the specific MDM implementation tasks within an organization.

8. Conclusion

In the current paper, we propose a functional model of an MDM solution intended for development within the iterative strategy. The model is designed for the early stages of the MDM implementation, and its main purpose is to translate the needs of an organization into the MDM terminology in order to estimate the percentage of MDM specific tasks. It allows to plan and estimate necessary functionality of the MDM solution and proceed with the tools selection and composing technical requirements. We provide a case study of the real MDM projects that were implemented with the model in question.

As a further direction of our research, we plan to develop evaluation techniques for the early-stage MDM project functionality and fine-grained metrics of the MDM solution complexity. Additionally, we intend to map the functionality of a typical MDM solution to various MDM products, as well as integrate our approach more tightly with knowledge management [23–24] and visualization techniques [25–26].

References

- [1]. Khatri V., Brown C.V. Designing data governance. *Communications of the ACM*, 53(1), 2010, pp. 148–152.
- [2]. DAMA-DMBOK: Data Management Body of Knowledge, 2017. 588 p.
- [3]. Andrichenko A.N. Tendencies and condition in the field of reference data management in the engineering industry. *Ontology of designing*, 2 (4), 2012, pp. 25–35. (in Russian).
- [4]. Nemtsov E.F. Reference data in the intelligent railway transport management system. *Automation, communication and Informatic*, 2, 2020, pp. 15–18. (in Russian).
- [5]. Golubev S.S., Lotsmanov A.N., Kuzin A. Y., Soloviev V.G., Kozlov A.D., Grigoriev B.A. The branch system of the National Standard Reference Data Service for the Oil and Gas Complex. *Legislative and applied metrology*, 3, 2020, pp. 12–16. (in Russian).
- [6]. Yanchenko G.A. On the standard reference data density properties of rocks. *Mining information and analytical bulletin*, 8, 2011, pp. 111–115. (in Russian).
- [7]. Chigirinsky Y., L. Method of reliability improvement of reference data. *Izvestia Volgogradskogo universitets*, 13 (86), 2011, pp. 55–61. (in Russian).
- [8]. Gartner Glossary, <https://www.gartner.com/en/glossary>, accessed 30.02.2025.
- [9]. Walker S. Dayley A., Parker S., Hawker M. Magic Quadrant for Master Data Management. Gartner, 2021.
- [10]. Zmud R.W. An Examination of ‘Push-Pull’ Theory Applied to Process Innovation in Knowledge Work. *Management Science*, 30 (6), 1984, pp. 727–738.
- [11]. Ladley J. Data Governance: How to Design, Deploy, and Sustain an Effective Data Governance Program. Academic Press; 2nd edition, 2019. 350 p.
- [12]. Kuznetsov S.V., Koznov D.V. Master data management in an iterative approach. *Ontology of designing*, 11(2), 2021, pp. 170–184. (in Russian).
- [13]. Silvola R., Jääskeläinen O., Kropsu-Vehkaperä H., Haapasalo H. Managing one master data – Challenges and preconditions. *Industrial Management & Data Systems*, 111(1), 2011, pp. 146–162.

- [14]. Yourdon E., Constantine L.L. Structured Design: Fundamentals of a Discipline of Program and Systems Design. Yourdon Press, 1975. 348 p.
- [15]. Jacobson I. Object-Oriented Software Engineering. ASM Press, 1992. 528 p.
- [16]. Ould M.A. Business Processes: Modelling and Analysis for Re-Engineering and Improvement. Wiley, 1995.
- [17]. Vassiliadis P., Simitsis A., Skiadopoulos S. Conceptual modeling for ETL processes. DOLAP, 2002, pp. 14–21.
- [18]. CMMI Data Management Maturity Model (DMM). CMMI Institute (website). <http://bit.ly/1Vev9xx>, accessed 30.01.2025.
- [19]. IBM Data Governance Council Maturity Model <https://ibm.co/2sRfBIn>, accessed 30.01.2025.
- [20]. Velocity Methodology. Best Practices. Informatica. 2008.
- [21]. O'Kane B., Moran M. P. The Seven Building Blocks of MDM: A Framework for Success. August 2016. Gartner. ID: G00311161.
- [22]. Ofner M.H., Straub K., Otto B., Österle H. Management of the master data lifecycle: a framework for analysis. J. Enterp. Inf. Manag, 26(4), 2013, pp. 472–491.
- [23]. Gavrilova T.A., Kudryatsev D. V. Knowledge management: from words to business. Intelligent Enterprise: RE, 12–13 (101), 2004, pp. 48. (in Russian).
- [24]. Gavrilova T.A. Knowledge presentation in expert system ABTANTECT. Izvestia Acadimii Nauk USSR, 5, 1984, pp. 165–173. (in Russian).
- [25]. Koznov D.V., Peregodov A.F., et al. Visual environment for designing broadcast software. Sistemnoye programmirovaniye, 2 (1), 2006, pp. 142–168. (in Russian).
- [26]. Ivanov A., Koznov D., et al. Behavior model RTST++. Zapiski seminarov kafedry sistemnogo programmirovaniya "Case-tool RTST++", 1, 1998, pp. 37–52. (in Russian).

Информация об авторах / Information about authors

Сергей Викторович КУЗНЕЦОВ – преподаватель кафедры прикладной кибернетики математико-механического факультета СПбГУ, исполнительный директор ООО «Юнидата». Сфера научных интересов: управление данными, управление мастер-данными, алгоритмы на графах, глубокое машинное обучение.

Sergey Viktorovich KUZNETSOV – a lecture of St.Petersburg University (Applied Cybernetics Chair), CEO of Unidata Ltd since 2014. Research interests: data government, master data management, graph algorithms, deep learning.

Дмитрий Владимирович КОЗНОВ, доктор технических наук, профессор кафедры системного программирования математико-механического факультета СПбГУ. Научные интересы: программная инженерия, модельно-ориентированная разработка программного обеспечения и dsls (в частности, для enterprise-приложений и телекоммуникаций), машинное обучение и нейросети в программной инженерии, техническая документация.

Dmitry Vladimirovich KOZNOV – Dr. Sci. (Tech.), Professor of the System Programming Department of St. Petersburg State University. Research interests: software engineering, model-driven software development & dsl (in particular, for enterprise applications and telecom), machine learning and neural networks for software data, technical documentation.

Дмитрий Вадимович ЛУЦИВ – кандидат физико-математических наук, доцент кафедры системного программирования Санкт-Петербургского государственного университета. Область научных интересов: программная инженерия, анализ данных программных проектов, анализ документации, системное программирование.

Dmitry Vadimovich LUCIV – PhD in computer science, associate professor of System Programming Department at Saint Petersburg State University, Russia. Research interests: software engineering, software data analysis, documentation analysis, systems programming.



DOI: 10.15514/ISPRAS-2025-37(4)-10

MaxSMT-решатель, поддерживающий режим портфолио

¹ В.В. Фомина, ORCID: 0009-0003-7132-722X <viki.fomina@yandex.ru>

² В. Соболев, ORCID: 0000-0002-2371-225X <vsobol@itmo.ru>

¹ Д.В. Кознов, ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

¹ Санкт-Петербургский государственный университет,
Россия, 199034, г. Санкт-Петербург, Университетская наб., д. 7–9.

² Университет ИТМО,
Россия, 197101, г. Санкт-Петербург, Кронверкский пр., д. 49, лит. А.

Аннотация. Задача MaxSMT заключается в определении выполнимости формулы теории первого порядка с ограничениями. В статье представлен подход к решению этой задачи для случая бескванторных теорий, которые обычно разрешимы и востребованы на практике для генерации тестов к программному обеспечению. Подход включает в себя модифицированный MaxSAT-алгоритм PrimalDualMaxRes, в котором использование SAT-решателя заменено SMT-решателем, а также реализацию режима портфолио. Последний означает, что итоговый MaxSMT-алгоритм запускается параллельно с несколькими SMT-решателями, и в качестве конечного результата берется наилучший. В качестве SMT-решателей, на базе которых построен режим портфолио, были выбраны Z3, Yices и Bitwuzla. Подход реализован с использованием открытой Kotlin-библиотеки KSMT, реализующей инфраструктуру для работы с набором MaxSMT-решателей. В статье также представлен первый тестовый набор (benchmark) для MaxSMT-задачи. Проведенные эксперименты подтверждают эффективность предложенного решения. Разработанное портфолио MaxSMT-решателей справляется с примерами из тестового набора данных более, чем в четыре раза быстрее существующего MaxSMT-решателя vZ (проект Z3).

Ключевые слова: задача MaxSMT; MaxSMT-решатель; библиотека KSMT; SMT-решатель; режим портфолио.

Для цитирования: Фомина В.В., Соболев В., Кознов Д.В. MaxSMT-решатель, поддерживающий режим портфолио. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 177–188. DOI: 10.15514/ISPRAS-2025-37(4)-10.

MaxSMT Solver with Portfolio Mode Support

¹ V.V. Fomina ORCID: 0009-0003-7132-722X <viki.fomina@yandex.ru>

² V. Sobol ORCID: 0000-0002-2371-225X <vsobol@itmo.ru>

¹ D.V. Koznov ORCID: 0000-0003-2632-3193 <d.koznov@spbu.ru>

¹ Saint-Petersburg State University,
7/9 Universitetskaya emb., St. Petersburg, 199034, Russia.

² ITMO University,
Kronverksky Pr. 49, bldg. A, St. Petersburg, 197101, Russia.

Abstract. The MaxSMT problem is to determine the satisfiability of a first-order theory formula with constraints. The paper presents an approach to solving this problem for the case of quantifier-free theories, which are usually decidable and widely applicable in practice for software test generation. The approach includes a modified MaxSAT algorithm PrimalDualMaxRes, in which the use of a SAT solver is replaced by an SMT solver, as well as the implementation of the portfolio mode. The latter means that the final MaxSMT algorithm is run in parallel with several MaxSMT solvers, and the best one is taken as the result. Z3, Yices and Bitwuzla were chosen as the SMT solvers on which the portfolio mode is built. The approach is implemented using the open-source Kotlin library KSMT, which implements the infrastructure for working with a set of MaxSMT solvers. The paper also presents the first benchmark for the MaxSMT problem. The evaluation demonstrates the competitiveness of the proposed solution. The developed portfolio of MaxSMT solvers outperforms the existing MaxSMT solver vZ (Z3 project), solving instances from the benchmark more than four times faster.

Keywords: MaxSMT; MaxSMT solver; KSMT; SMT solver; portfolio.

For citation: Fomina V.V., Sobol V., Koznov D.V. MaxSMT solver with portfolio mode support. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025. pp. 177-188 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-10.

1. Введение

В области анализа программ широко известна задача SAT (Boolean Satisfiability Problem), определяющая выполнимость булевых формул [1-2]. Эффективное решение этой задачи, известное как SAT-революция, позволило значительно повысить точность и масштабируемость автоматической верификации и тестирования программного обеспечения [1]. Обобщением этой задачи на исчисление предикатов первого порядка является задача SMT (Satisfiability Modulo Theories) [3-4]. Она позволяет проверять выполнимость формул, описывающих более широкое множество свойств программ, чем булевы формулы. В свете этого создаются различные теории первого порядка для формального описания различных структур данных – массивов [5], битовых векторов [5] и пр., что позволяет формализовывать и автоматически проверять сложное поведение программ. Активно используются бескванторные фрагменты таких теорий как битовые вектора, неинтерпретированные функции, числа с плавающей точкой, линейная целочисленная и рациональная арифметики. Исключение утверждений с кванторами гарантирует разрешимость перечисленных теорий, что позволяет определять выполнимость или невыполнимость поведения программ. При этом сохраняется возможность описывать широкий диапазон свойств программ. Для автоматизации проверок этих свойств используются SMT-решатели – Z3 [6], Yices [7], Bitwuzla [8] и др.

При практическом использовании SMT-решателей часто возникает необходимость гибко управлять формулами, которые передаются в решатель [9]. В частности, важно иметь возможность задавать различные приоритеты для проверки отдельных частей формулы, например, в случае, когда требуется определить выполнимость не всей формулы. Такая

задача называется MaxSMT [9-14] (аналогичная задача для SAT называется MaxSAT [15]). Задача MaxSMT предполагает многократное решение задачи SMT.

Формально, задача MaxSMT описывается так. Пусть в рамках некоторой теории первого порядка имеется два набора утверждений H и S . Для набора H ставим задачу SMT. Далее каждому утверждению s из набора S ставим в соответствие приоритет его выполнимости – вес $weight(s)$. Решить задачу MaxSMT означает найти набор утверждений K из всех выполнимых утверждений, входящих в S , для которого значение $\sum_{s \in K} weight(s)$ является наибольшим из возможных. Условием наличия решения задачи является выполнимость всех утверждений из H .

Задача MaxSMT может использоваться при автоматической генерации тестов с применением символьного исполнения [16-17]. Применение MaxSMT-решателя совместно с символьной машиной позволяет создавать тесты, которые удовлетворяют определенным условиям пути, т.е. попадают в нужное место программы. Кроме того, в таких тестах выполняются дополнительные ограничения, не противоречащие условиям пути. Например, можно задать диапазоны для размеров генерируемых массивов. Подобные ограничения, в частности, позволяют управлять размерами создаваемых в программе объектов, что помогает избегать чрезмерного использования памяти.

Существующие исследования, посвященные MaxSMT [18-21], не рассматривают практические подходы к повышению скорости решения задачи, в то время как такие подходы могут позволить повысить применимость MaxSMT-решателей, в том числе для анализа программ, который требует решения задачи MaxSMT. Последний можно ускорить применением режима портфолио. Этот режим предполагает одновременный запуск различных MaxSMT-решателей на одной задаче с целью использовать решение того решателя, который показал лучший результат за фиксированное время.

Данная работа посвящена эффективному решению задачи MaxSMT и рассматривает как теоретические, так и практические подходы к ее решению.

В работе используется известный алгоритм решения задачи MaxSAT – PrimalDualMaxRes [18], в котором шаг, позволяющий получать промежуточные решения для задачи MaxSAT, заменен шагом, позволяющим получать промежуточные решения для более общей задачи MaxSMT. Это обеспечивает возможность по истечении выделенного лимита времени вернуть наилучшее вычисленное решение (субоптимальное), даже если оно не является оптимальным. В качестве способа взаимодействия с SMT-решателями используется проект KSMT [22], который предоставляет универсальный интерфейс работы с различными SMT-решателями. Это позволяет реализовать решение задачи MaxSMT с использованием всех доступных в библиотеке SMT-решателей. В рамках работы создан тестовый набор данных (benchmark) для задачи MaxSMT [23].

Статья организована следующим образом. В разделе 2 обосновывается выбор технологий для решения поставленной задачи. В разделе 3 выполнен краткий обзор предложенного решения. В разделе 4 приводится адаптация алгоритма PrimalDualMaxRes для решения задачи MaxSMT. В разделе 5 предлагается реализация режима портфолио. В разделе 6 описываются эксперименты. В разделе 7 выполнен обзор близких к данному исследованию работ.

2. Выбор технологий

Известные алгоритмы решения задачи MaxSMT имеют ряд ограничений. Некоторые из них не позволяют получать промежуточные решения [19], другие используют ограничения на кардинальность [20], которые поддерживаются далеко не каждым SMT-решателем, а третьи эффективны только для небольших задач [21]. Первое обстоятельство исключает возможность получить потенциально хорошее, но неоптимальное решение, если оптимальное решение не было найдено за разумное время. Второе обстоятельство не позволяет использовать большинство известных SMT-решателей для решения задачи

MaxSMT, тогда как согласно соревнованиям SMT-COMP [24] на разных теориях наилучшие результаты показывают разные SMT-решатели. Третье обстоятельство делает алгоритмы непригодными для производственных задач.

В качестве SMT-решателей, на базе которых построен режим портфолио для задачи MaxSMT, были выбраны Z3 [6], Yices [7] и Bitwuzla [8]. Эти проекты давно разрабатываются, предлагают поддержку широкого набора теорий, а также демонстрируют хорошую производительность на соревнованиях SMT-COMP.

Режим портфолио реализован в рамках Kotlin-библиотеки KSMT [22], предоставляющей универсальный программный интерфейс для доступа к набору SMT-решателей. Эта библиотека позволяет реализовать алгоритм решения задачи MaxSMT один раз и использовать его с разными SMT-решателями. Кроме того, проект KSMT реализует конвертеры формул из конкретного представления SMT-решателя в универсальный формат библиотеки, что упрощает реализацию режима портфолио задачи MaxSMT.

В качестве базового алгоритма использован MaxSAT-алгоритм PrimalDualMaxRes. Он состоит из двух шагов: шаг получения промежуточного решения и шаг преобразования утверждений с учетом промежуточного решения с целью найти лучшие решения на следующих итерациях. Эти шаги повторяются до тех пор, пока не будет найдено оптимальное решение. Алгоритм PrimalDualMaxRes способен выдавать субоптимальные решения в условиях ограниченного времени. Также он отличается высокой скоростью сходимости. Целесообразность его использования в данном исследовании связана с возможностью замены вызовов SAT-решателей на вызовы SMT-решателей, сохранив все те преимущества, которые обозначены выше, и применив их для задачи MaxSMT.

3. Обзор решения

На рис. 1 представлена схема взаимодействия между разработанным MaxSMT-решателем и используемыми SMT-решателями. Формулы, специфичные для конкретного SMT-решателя, преобразуются в универсальный формат библиотеки KSMT, с которым работает MaxSMT-решатель. Решатель задачи MaxSMT позволяет устанавливать ограничения на время решения задачи, при этом полученные решения содержат информацию о том, являются ли они оптимальными. MaxSMT-решатель может быть сконфигурирован так, чтобы задача MaxSMT решалась с использованием любого из SMT-решателей, поддерживаемых KSMT. Решатель реализует адаптированную под решение задачи MaxSMT версию алгоритма PrimalDualMaxRes. MaxSMT-решатель может быть использован для работы в режиме портфолио.

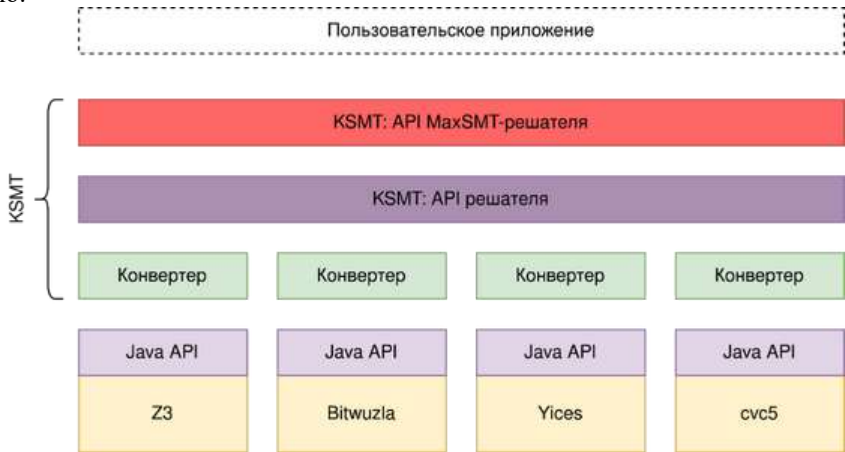


Рис. 1. Взаимодействие MaxSMT-решателя с используемыми SMT-решателями.
Fig. 1. Interaction of the MaxSMT solver with the used SMT solvers.

4. Адаптация алгоритма PrimalDualMaxRes

Поскольку процесс построения промежуточных решений в оригинальном алгоритме PrimalDualMaxRes ограничен работой с булевыми формулами, этот шаг заменен на семантически аналогичный, но позволяющий работать с формулами теорий первого порядка. Опишем, как это делается.

Пусть даны множества $H = \{h_1, h_2, \dots, h_n\}$ и $S = \{s_1, s_2, \dots, s_n\}$, которые включают в себя утверждения в контексте решения задачи MaxSMT – без весов и с весами соответственно. Пусть также имеется множество SLS для хранения решений, найденных к текущему моменту. На этом множестве задано отношение частичного порядка, определяемое суммой весов выполнимых утверждений. Изначально $SLS = \emptyset$. Далее проверяется выполнимость формулы $F_1 \leftarrow \bigwedge_{h \in H} h \wedge (s_1 \wedge s_2)$.

В том случае, если эта формула невыполнима, происходит переход ко второму и последнему шагу первоначального алгоритма PrimalDualMaxRes.

При условии выполнимости формулы получено промежуточное решение M . Если $M > \sup SLS$, то элемент M добавляется в множество SLS . Далее проверяется выполнимость формулы $F_2 \leftarrow F_1 \wedge (s_1 \wedge s_2 \wedge \dots \wedge s_{k+2})$. Поиск оптимального числа утверждений k , добавляемых на каждом шаге, требует дальнейшего исследования. Шаг адаптированного алгоритма повторяется до тех пор, пока не будет получено невыполнимое множество утверждений или не станет ясно, что все утверждения из множества S выполнимы.

В первом случае осуществляется переход к следующему шагу исходного алгоритма PrimalDualMaxRes. Во втором выполнение модифицированного алгоритма завершается с найденным оптимальным решением, соответствующим $\sup SLS$.

5. Реализация режима портфолио

Представленное решение расширяет реализацию режима портфолио решателей в KSMT, изначально созданную для задачи SMT, ориентируясь на решение задачи MaxSMT. Портфолио MaxSMT-решателей управляется запускаящим его процессом. Каждый MaxSMT-решатель, используемый в портфолио, работает в отдельном процессе – см. рис. 2. Обсудим детали реализации решения. Для управления портфолио SMT-решателей KSMT реализует класс PortfolioSolver. Этот класс обеспечивает выполнение набора стандартных операций (добавление утверждений, проверка их выполнимости и т.д.) для каждого решателя из портфолио в порядке, заданном пользователем. В KSMT были внесены изменения для того, чтобы добавленная операция решения MaxSMT не нарушала этот порядок (MaxSMT-решатель в KSMT наследует операции SMT-решателя). В класс PortfolioSolver также добавлена стратегия обработки решений: если один из решателей возвращает оптимальное решение, оно принимается в качестве ответа; в противном случае возвращается решение от того решателя, который нашел наилучшее решение за отведенное время.



Рис. 2. Работа MaxSMT-решателя в режиме портфолио.

Fig. 2. MaxSMT solver operation in portfolio mode.

Объект класса SolverRunner хранит состояние решателя в процессе, который управляет портфолио. Это позволяет пересоздать процесс решателя и восстановить его состояние в случае зависания или неожиданного завершения. В библиотеку KSMT был также добавлен механизм восстановления утверждений с весами, чтобы обеспечить возможность восстановления состояния MaxSMT-решателя.

Объект класса SolverRunner перенаправляет запросы на выполнение операций экземпляру класса SolverExecutor, который затем направляет их по протоколу RPC в процесс решения задачи MaxSMT. В свою очередь, класс SolverWorkerProcess описывает действия, выполняемые в процессе решения задачи MaxSMT.

6. Эксперименты

Эксперименты осуществлялись на агентах сервиса GitHub [25] с процессором Intel Xeon E5, 2.6 GHz (4 ядра) и 16 Гб RAM, под управлением Linux (дистрибутив Ubuntu 22.04).

Поскольку в сообществе отсутствовал тестовый набор данных (benchmark) для задачи MaxSMT, то он был разработан в рамках нашего исследования. Это было сделано следующим образом. Сначала был взят инкрементальный тестовый набор данных 2023 года для задачи SMT [26]. Затем этот набор был преобразован для решения задачи MaxSMT путем назначения весов некоторым утверждениям. Веса назначались утверждениям, добавленным после выполнения первой операции push. Наконец, при помощи MaxSMT-решателя vZ [21] (часть проекта Z3), работавшего с ограничением времени в пять минут, были сгенерированы оптимальные решения задачи MaxSMT для примеров из преобразованного инкрементального тестового набора данных. Таким образом, получившийся тестовый набор включает примеры с бескванторными фрагментами следующих теорий: массивов, битовых векторов, неинтерпретированных функций, чисел с плавающей точкой, линейной целочисленной и рациональной арифметик, а также различные комбинации этих теорий.

Эксперименты проводились для пяти следующих MaxSMT-решателей.

- Существующий MaxSMT-решатель vZ в стандартной конфигурации.
- Наш подход, работающий на каждом из SMT-решателей по отдельности – Z3, Yices и Bitwuzla.
- Наш подход, работающий в режиме портфолио с решателями Z3, Yices и Bitwuzla.

Представительность этого набора MaxSMT-решателей связана с тем, что входящие в него решатели основаны на производительных SMT-решателях и реализуют алгоритмы решения задачи MaxSMT с высокой скоростью сходимости.

В качестве метрик оценки качества решения было взято время работы, а также полученная оценка – значение из отрезка от нуля до единицы, вычисляемое как отношение суммы весов выполнимых утверждений, найденной решателем за отведенный промежуток времени, к наибольшей возможной сумме весов выполнимых утверждений. Было поставлено два эксперимента: в оптимальном и субоптимальном режимах. Целью первого эксперимента было оценить, насколько успешно представленный в работе MaxSMT-решатель справляется с набором примеров по сравнению с существующими решениями, а также проверить, может ли режим портфолио обеспечить заметный прирост производительности при работе с различными теориями. Целью второго эксперимента было убедиться, что если в условиях ограниченного времени MaxSMT-решатель не успевает найти оптимальное решение, то он может вернуть промежуточное, которое зачастую может оказаться близким к оптимальному.

В рамках каждого эксперимента было сделано по семь запусков. Все графики строились по средним значениям. Время работы всех решателей измерялось в том же процессе, в котором они запускались. Для решателя, работающего в режиме портфолио, время измерялось процессом, который этим портфолио управлял. Это позволило учесть затраты времени на взаимодействие между процессом-координатором и процессом решения задачи MaxSMT.

6.1 Оптимальный режим

В оптимальном режиме задача считается успешно решенной только в том случае, если найдено оптимальное решение.

В этом режиме на решение примера устанавливалось ограничение в 60 секунд. По результатам запусков построен график, который отражает зависимость количества примеров от времени, не дольше чем за которое каждый из них был решен – см. график, представленный на рис. 3. Стоит отметить, что для каждого примера было вычислено лучшее время решения, полученное в рамках запуска решателей. Именно эти значения использовались при построении функции, обозначенной на графике как «Лучшее».

Анализ графика с рис. 3 показывает, что решатель *vZ* смог решить около двухсот таких примеров, что на каждый из них он затратил менее пяти миллисекунд. В то же время у остальных решателей либо отсутствуют примеры, решенные менее чем за пять миллисекунд, либо их всего несколько. Для других решателей отметка в двести примеров достигается при значении времени в десять миллисекунд, тогда как *vZ* за это время решает уже триста примеров. Это можно объяснить тем, что исходная реализация MaxSMT-решателя проекта Z3 (*vZ*) не расходует ресурсы на преобразование формул в формат библиотеки KSMT, а также на взаимодействие с другими процессами, как это делает портфолио решателей.

Можно заметить, что начиная с какого-то момента, с ростом времени количество успешно решенных примеров у решателя, работающего в режиме портфолио, начинает увеличиваться быстрее, чем у остальных решателей. Это обусловлено тем, что накладные расходы на запуск портфолио оказываются незначительными по сравнению с выигрышем во времени, который он дает. График на рис. 3. демонстрирует, что функция, отражающая результаты измерений для портфолио решателей, наиболее близко соответствует функции, отражающей лучший результат.

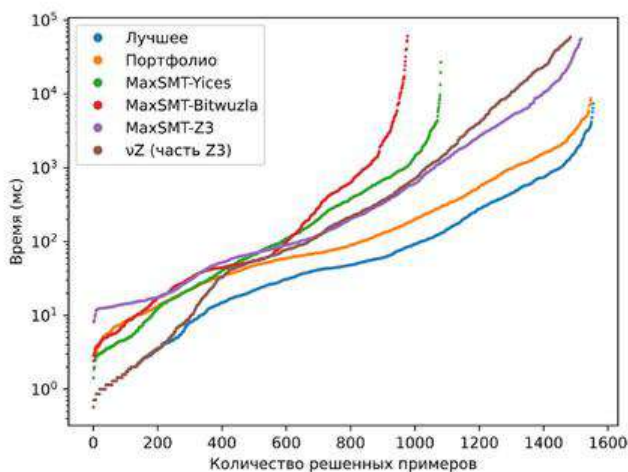


Рис. 3. Режим портфолио: среднее количество решенных примеров не дольше, чем за время (оптимальный режим).

Fig. 3. Portfolio mode: the average number of solved examples is no longer than in time (optimal mode).

В ходе проведенного эксперимента также было установлено количество успешно решенных примеров для каждого из MaxSMT-решателей. Кроме этого, было рассчитано среднее время, затраченное на решение каждого примера (см. табл. 1).

Табл. 1 демонстрирует, что решатель в режиме портфолио превосходит другие MaxSMT-решатели по количеству успешно решенных примеров. Меньше всего примеров решено

MaxSMT-решателем, использующим SMT-решатель Bitwuzla. Это связано с тем, что Bitwuzla, как и Yices, поддерживает не все теории из тестового набора данных. Согласно данным из табл. 1, лучшее среднее время работы демонстрирует решатель на базе Yices. Портфолио MaxSMT-решателей несколько уступает по этому показателю. Однако решатель в режиме портфолио успешно справляется с гораздо большим набором задач. Он также успешно справляется с большим числом примеров, чем νZ , при этом в восемь раз превосходя его по среднему времени решения примера. В то же время среднее время работы νZ заметно превышает время работы MaxSMT-решателя, использующего решатель Z3 библиотеки KSMT. Это может быть обусловлено наличием неоднозначных операций над unsat-ядрами [27], которые выполняются в рамках нативной реализации Z3. Эти операции позволяют получить выигрыш в производительности на каких-то примерах, но в то же время на других примерах ведут к деградации.

Табл. 1. Среднее количество решенных примеров (оптимальный режим).

Table 1. Average number of solved examples (optimal mode).

Решатели	Кол-во решенных примеров	Среднее время на пример (мс)
Портфолио решателей	1546 ± 2	495 ± 11
Yices	1080 ± 2	459 ± 24
Bitwuzla	943 ± 46	883 ± 139
Z3	1508 ± 4	2582 ± 57
νZ (часть Z3)	1479 ± 3	3987 ± 40

6.2 Субоптимальный режим

В субоптимальном режиме наш подход учитывает любое решение, даже если оно не является оптимальным. При этом в наших экспериментах на решение примера выделялось 10 секунд. В результате запусков построен график, представленный на рис. 4, который иллюстрирует минимальное количество примеров, на которых полученная оценка не меньше соответствующего значения. На графике отражены показатели для различных MaxSMT-решателей, а также лучший результат по каждому из примеров. Все решатели смогли набрать на девятистах примерах максимальные оценки, и поэтому эти примеры не отражены на графике. График показывает, что решатель в режиме портфолио справился с наибольшим количеством примеров, набрав максимальные оценки. И только на нескольких примерах он получил оценки, отличные от единицы, однако близкие к ней. Этот показатель близок к лучшему из возможных. MaxSMT-решатель, использующий SMT-решатель Yices, тоже практически на всех примерах набрал оценки, равные единице. Однако, поскольку Yices не поддерживает некоторые теории, он запустился не на всех примерах. Остальные решатели на многих примерах получают оценки, отличные от единицы, причем среди этих значений немало таких, которые значительно отличаются от единицы.

В ходе проведенного эксперимента также была вычислена сумма оценок, набранных решателем на тестовом наборе данных, и среднее время решения каждого примера (табл. 2).

Табл. 2. Средние сумма оценок и время, затраченное на пример (субоптимальный режим).

Table 2. Average scores and time spent on the example (suboptimal mode).

Решатели	Оценки	Среднее время на пример (мс)
Портфолио решателей	1555 ± 1	522 ± 9
Yices	1088 ± 1	426 ± 10
Bitwuzla	979 ± 3	727 ± 32
Z3	1505 ± 3	1916 ± 27
νZ (часть Z3)	1473 ± 3	2451 ± 12

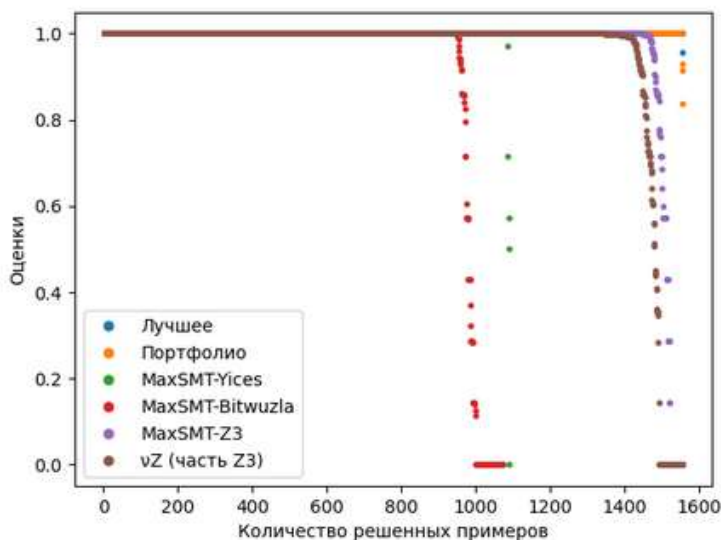


Рис. 4. Распределение оценок в зависимости от количества примеров (субоптимальный режим)

Fig. 4. Distribution of the score depending on the number of examples (suboptimal mode)

Лучшее среднее время работы демонстрирует решатель на базе Yices. Тем не менее, решатель в режиме портфолио, получает лучшие оценки на тестовом наборе данных, хоть и при несколько большем среднем времени работы, чем Yices. Портфолио решателей справляется с примерами в четыре раза быстрее, чем vZ, и при этом набирает лучшие оценки на тестовом наборе данных.

7. Обзор существующих работ

К настоящему времени в сообществе не представлено ни одного MaxSMT-решателя, который бы поддерживал режим портфолио. Более того, в современных решателях реализация решения задачи MaxSMT тесно связана с конкретным SMT-решателем. Это лишает возможности выбрать наиболее подходящий для конкретной задачи SMT-решатель, что заметно снижает эффективность, а значит и применимость MaxSMT-решателей.

Одним из способов преодоления этого ограничения является использование универсальных интерфейсов (так называемых «обертки») для взаимодействия с различными SMT-решателями. К таким инструментам относятся metaSMT [28], JavaSMT [29], pySMT [30]. Они позволяют абстрагироваться от особенностей реализации конкретных решателей, что, в теории, позволяет реализовать алгоритм MaxSMT на более высоком уровне и использовать в процессе решения задачи любой поддерживаемый «оберткой» SMT-решатель. Однако на практике существующие инструменты пока не используют эту возможность.

Наиболее эффективным существующим MaxSMT-решателем является vZ. Этот решатель может выдавать не только оптимальные решения, но и промежуточные. Одним из ключевых преимуществ vZ является способность работать с большим набором теорий, включая такие популярные теории, как битовые вектора, массивы, линейная целочисленная и рациональная арифметики, неинтерпретированные функции, а также числа с плавающей точкой. Тем не менее, vZ ограничен использованием SMT-решателя Z3, что сужает его потенциальную применимость в случаях, где использование других SMT-решателей могло бы позволить показать лучшие результаты.

Стоит отметить ОМТ-решатель OptiMathSAT [31], основанный на SMT-решателе MathSAT [32] с закрытым исходным кодом. OptiMathSAT представляет собой инструмент для решения задачи ОМТ (Optimization Modulo Theories), которая заключается в поиске оптимального решения SMT-проблемы относительно целевых функций, описывающих ключевые показатели системы. Задача MaxSMT, для которой нетрудно определить целевую функцию, может быть сведена к задаче ОМТ [31]. Однако такой способ не является эффективным способом решения задачи MaxSMT, поскольку оптимизационный механизм ОМТ-решателя может выполнять избыточный перебор различных комбинаций, приводящих к одинаковым результатам [33].

8. Заключение

В данной работе представлена адаптированная версия алгоритма PrimalDualMaxRes для решения задачи MaxSAT, которая позволяет решать более общую задачу – MaxSMT. Модифицированный алгоритм реализован с использованием Kotlin-библиотеки KSMT, поддерживая возможность работы в режиме портфолио.

Эксперименты продемонстрировали, что использование режима портфолио позволяет решать большинство примеров быстрее. Также установлено, что на тестовом наборе данных портфолио MaxSMT-решателей превосходит по времени решения существующий MaxSMT-решатель vZ (проект Z3) не менее чем в четыре раза, демонстрируя мощность предложенного решения. В качестве развития данного исследования можно отметить возможность интеграции итогового MaxSMT-решателя для генерации тестов в специализированные среды разработки – встроенных систем реального времени [34], систем телевидения [35] и др.

Список литературы / References

- [1]. Alouneh, S., Abed, S. E., Al Shayeji, M. H., Mesleh, R. A comprehensive study and analysis on SAT-solvers: advances, usages and achievements. *Artificial Intelligence Review*, vol. 52, 2019, pp. 2575–2601.
- [2]. Misonizhnik, A., Mordvinov, D. On Satisfiability of Nominal Subtyping with Variance // *Proceedings of 33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, 2019.
- [3]. Barrett, C., Sebastiani, R., Seshia, S. A., Tinelli, C. Satisfiability modulo theories // *Handbook of satisfiability*. IOS Press, 2021, pp. 1267–1329.
- [4]. Костюков, Ю. О., Батоев, К. А., Мордвинов, Д. А., Костицын, М. П., Мисонизник, А. В. Автоматическое доказательство корректности программ с динамической памятью. *Труды ИСП РАН*, том 31, вып. 5, 2019 г., стр. 37–62.
- [5]. Barrett, C., Tinelli, C., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y. Satisfiability Modulo Theories: A Beginner's Tutorial. *International Symposium on Formal Methods*, 2024, pp. 571–596.
- [6]. De Moura, L., Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, March, pp. 337–340.
- [7]. Dutertre, B. Yices 2.2. In *International Conference on Computer Aided Verification*, 2014, pp. 737–744.
- [8]. Niemetz, A., Preiner, M. Bitwuzla. In *International Conference on Computer Aided Verification*, 2023, pp. 3–17.
- [9]. Pasareanu, C. S., Phan, Q. S., Malacaria, P. Multi-run side-channel analysis using Symbolic Execution and Max-SMT // *Proceedings of IEEE 29th Computer Security Foundations Symposium (CSF)*. 2016, pp. 387–400.
- [10]. Brockschmidt, M., Larra, D., Oliveras, A., Rodriguez-Carbonell, E., Rubio, A. Compositional safety verification with Max-SMT // *Formal Methods in Computer-Aided Design (FMCAD)*. 2015, pp. 33–40.
- [11]. Terra-Neves, M., Machado, N., Lynce, I., Manquinho, V. Concurrency debugging with MaxSMT // *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, No. 01, 2019, pp. 1608–1616.
- [12]. Larraz, D., Nimkar, K., Oliveras, A., Rodriguez-Carbonell, E., Rubio, A. Proving non-termination using Max-SMT // *Proceedings of Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* 26. Springer, 2014, pp. 779–796.

- [13]. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A. Proving termination of imperative programs using Max-SMT. In 2013 Formal Methods in Computer-Aided Design, 2013, pp. 218–225.
- [14]. Albert, E., Gordillo, P., Rubio, A., Schett, M. A. Synthesis of super-optimized smart contracts using max-smt. In International conference on computer aided verification, 2020, pp. 177–200.
- [15]. Li, C. M., Manya, F. MaxSAT, hard and soft constraints. In Handbook of satisfiability. IOS Press, 2021, pp. 903–927.
- [16]. Мисонижник А.В., Бабушкин А.А., Морозов С.А., Костюков Ю.О., Мордвинов Д.А., Кознов Д.В. Автоматическое тестирование LLVM-программ со сложными входными структурами данных. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 49–62.
- [17]. Chen, T., Zhang, X. S., Guo, S. Z., Li, H. Y., Wu, Y. State of the art: Dynamic symbolic execution for automated test generation. Future Generation Computer Systems, vol. 29(7), 2013, pp. 1758–1773.
- [18]. Björner, N., Narodytska, N. Maximum satisfiability using cores and correction sets // Proceedings of Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015, pp. 246–252.
- [19]. Narodytska, N., Bacchus, F. Maximum satisfiability using core-guided MaxSAT resolution // Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28, No. 1, 2014.
- [20]. Morgado, A., Dodaro, C., Marques-Silva, J. Core-Guided MaxSAT with Soft Cardinality Constraints. Principles and Practice of Constraint Programming, 2014, pp. 564–573.
- [21]. Björner, N., Phan, A. D., Fleckenstein, L. vz-an optimizing SMT solver // Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015, Proceedings 21, 2015, pp. 194–199.
- [22]. KSMT, <https://github.com/UnitTestBot/ksmt> (дата обращения: 01.02.2025).
- [23]. MaxSMT benchmark, <https://github.com/viktoriia-fomina/ksmt/releases/tag/0.0.0> (дата обращения: 01.02.2025).
- [24]. SMT-COMP 2023, <https://smt-comp.github.io/2023/> (дата обращения: 01.02.2025).
- [25]. GitHub hosted runners, <https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners> (дата обращения: 01.02.2025).
- [26]. Тестовый набор данных для задачи SMT, <https://smt-lib.org/benchmarks.shtml> (дата обращения: 01.02.2025).
- [27]. Guthmann, O., Strichman, O., Trostanetski, A. Minimal unsatisfiable core extraction for SMT. In 2016 Formal Methods in Computer-Aided Design (FMCAD), 2016, pp. 57–64.
- [28]. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R. metaSMT: Focus on Your Application not on Solver Integration. 2011, p. 47.
- [29]. Baier, D., Beyer, D., Friedberger, K. JavaSMT 3: Interacting with SMT solvers in Java // Proceedings of International Conference on Computer Aided Verification, 2021, pp. 195–208. DOI: 10.1007/978-3-030-81688-9_9.
- [30]. Gario, M., Micheli, A. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms // Proceedings of SMT workshop, 2015.
- [31]. Sebastiani, R., Trentin, P. OptiMathSAT: A tool for optimization modulo theories. Journal of Automated Reasoning, 64(3), 2020, pp. 423–460.
- [32]. Cimatti, A., Griggio, A., Schaafsma, B. J., Sebastiani, R. The mathsat5 smt solver. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2013, pp. 93–107.
- [33]. Sebastiani, R., Trentin, P. On optimization modulo theories, MaxSMT and sorting networks // Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2017, pp. 231–248.
- [34]. Terekhov A.N., Romanovskii K.Y., Koznov D.V., Dolgov P.S., Ivanov A.N. RTST++: Methodology and a CASE tool for the development of information systems and software for real-time systems. Programming and Computer Software, 1999, 25 (5), pp. 276–281.
- [35]. Кознов Д.В., Перегудов А.Ф., Бугайченко Д.Ю., Чернячик Р.И., Казакова А.С., Павлинов А.А. Визуальная среда проектирования систем телевизионного вещания. Системное программирование, 2006, 2(1), стр. 142–168.

Информация об авторах / Information about authors

Виктория Викторовна ФОМИНА – аспирант математико-механического факультета СПбГУ. Сфера научных интересов: анализ программ.

Viktoriia Viktorovna FOMINA is a postgraduate student of the Faculty of Mathematics and Mechanics, St. Petersburg State University. Her research interests include program analysis.

Валентин СОБОЛЬ – выпускник Санкт-Петербургского Политехнического университета им. Петра Великого, преподаватель университета ИТМО. Научные интересы: статический анализ программ, символьное исполнение.

Valentyn SOBOL is a lecturer at ITMO University and a graduate of Peter the Great St. Petersburg Polytechnic University. His scientific interests include static analysis of programs and symbolic execution.

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования Санкт-Петербургского государственного университета, Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV – Dr. Sci. (Tech.), Associate Professor, Professor St. Petersburg State University (SPbSU). Research interests: software engineering, model-driven software development, program data, machine learning.

DOI: 10.15514/ISPRAS-2025-37(4)-11



Локализация целей в реальном времени на БПЛА с лазерным дальномером на подвесе

*В.Р. Саакян, ORCID:0000-0002-7552-4904 <vardan.sahakyan@student.rau.am>
В.Г. Мелконян, ORCID:0000-0002-3584-1294 <vahagn.melkonyan@student.rau.am>
С. Саргсян, ORCID: 0000-0002-8831-4965 <sevak.sargsyan@rau.am>*

*Российско-Армянский университет,
123, ул. Овсена Эмина, Ереван, 0051, Армения.*

Аннотация. В статье представлена новая система трехмерной локализации в реальном времени на базе БПЛА, которая сочетает монокулярную камеру, лазерный дальномер на стабилизирующем подвесе и бортовые алгоритмы компьютерного зрения. В отличие от предыдущих подходов, основанных на допущениях, таких как известный размер объекта или плоский рельеф местности, и часто проверяемых только в симуляции, наш метод обеспечивает точную локализацию целей без предварительных знаний об окружающей среде. Система осуществляет слежение и локализацию полностью на борту дрона за счёт активного слияния данных от датчиков и управления подвесом. Разработанный нами метод реализован в универсальном программном обеспечении и протестирован в полевых условиях, где продемонстрировал высокую точность и устойчивость.

Ключевые слова: беспилотный летательный аппарат (БПЛА); локализация объектов; монокулярная камера; лазерный дальномер; слияние датчиков; управление подвеской.

Для цитирования: Саакян В.Р., Мелконян В.Г., Саргсян С.С. Локализация целей в реальном времени на БПЛА с лазерным дальномером на подвесе. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 189–198. DOI: 10.15514/ISPRAS-2025-37(4)-11.

Благодарности: Работа выполнена при поддержке Комитета по науке Республики Армения (исследовательский проект № 23AA-1B007).

Real-Time Target Localization Using Gimbaled Laser on UAVs

V. Sahakyan, ORCID: 0000-0002-7552-4904 <vardan.sahakyan@student.rau.am>
V. Melkonyan, ORCID: 0000-0002-3584-1294 <vahagn.melkonyan@student.rau.am>
S. Sargsyan, ORCID: 0000-0002-8831-4965 <sevak.sargsyan@rau.am>

*Russian-Armenian University,
123, Hovsep Emin st., Yerevan, 0051, Armenia.*

Abstract. This paper presents a novel UAV-based system for real-time 3D-object localization that integrates a monocular camera, a gimbaled laser rangefinder, and an onboard computer vision. Unlike prior methods that rely on assumptions such as known object size, flat terrain, or simulation-only validation, our approach enables accurate localization of targets without requiring prior knowledge of the environment. The system performs real-time tracking and localization entirely onboard the drone through active sensor fusion and gimbal control. We implemented the method in a universal software framework and validated it through field experiments, demonstrating its accuracy and robustness in real-world conditions.

Keywords: UAV; object localization; real-time tracking; monocular camera; laser rangefinder; sensor fusion; gimbal control.

For citation: Sahakyan V.R., Melkonyan V.G., Sargsyan S.S. Real-Time Target Localization Using Gimbaled Laser on UAVs. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 189-198 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-11.

Acknowledgements. This work was supported by the Science Committee of RA (Research project № 23AA-1B007).

1. Введение

Беспилотные летательные аппараты (БПЛА) стали ключевой технологической платформой, применяемой в таких областях, как инспекция инфраструктуры, точное земледелие, мониторинг окружающей среды и поисково-спасательные операции (ПСО). Их способность работать на разной высоте, ориентироваться в сложных условиях и собирать визуальные данные высокого разрешения делает их подходящими для задач, требующих ситуационной осведомленности и оперативного реагирования. Одним из наиболее ценных возможностей в этих сценариях является локализация объектов в реальном времени - способность обнаруживать, отслеживать и точно определять трехмерное положение наземных объектов. Точная локализация объектов с помощью БПЛА необходима для выполнения таких задач, как мониторинг оборудования на удаленных инфраструктурных объектах, отслеживание диких животных или людей в рамках ПСО и управление сельскохозяйственными процессами. В этих сценариях простого визуального обнаружения или отслеживания объекта недостаточно: система должна также оценить координаты объекта относительно глобальной или локальной системы отсчета, чтобы запустить соответствующие последующие действия. Несмотря на прогресс в методах оценки глубины и локализации на основе компьютерного зрения, большинство существующих подходов опираются на ограничительные предположения - такие как известный размер объектов, плоский рельеф, статичность целей или наличие точных предварительных карт; и зачастую проверяются только в симуляторах или ограниченных средах. Более того, многие системы ориентированы либо на визуальное слежение, либо на оценку глубины, но лишь немногие из них обеспечивают интегрированное решение в реальном времени для полной трехмерной локализации в практических условиях применения.

Для решения вышеописанных задач мы предлагаем систему на базе БПЛА, которая сочетает в себе монокулярную камеру, лазерный дальномер на подвесе (gimbal) и бортовое компьютерное зрение для отслеживания и локализации как статичных, так и движущихся

целей в режиме реального времени. Предложенный подход устраняет необходимость в предварительном знании размеров объектов, структуры местности или внешней обработки, что позволяет гибко развертывать систему в разных средах. Система использует визуальное обнаружение для отслеживания целей, активно управляемый подвес для выравнивания лазерного луча по цели и лазерный дальномер для получения точных измерений расстояния, на основе которых определяются глобальные координаты.

Основные достижения данной работы:

- **Новая система на основе БПЛА**, которая сочетает монокулярную камеру, лазерный дальномер на стабилизирующей подвеске и бортовую обработку изображений для локализации объектов в реальном времени.
- **Полностью автономная работа в реальном времени** с активным управлением подвеской и объединением данных с датчиков, без необходимости предварительных знаний о размере объекта, рельефе или состоянии движения.
- **Экспериментальная проверка в условиях открытой местности**, демонстрирующая точную локализацию целей.

2. Обзор существующих методов

Точная локализация объектов с использованием БПЛА представляет собой сложную задачу, обусловленную ограничениями бортовых сенсоров, особенностями воздушной перспективы и требованиями к работе в реальном времени. Существующие подходы включают пассивные методы компьютерного зрения, технологии, учитывающие рельеф местности, и лазерное сканирование. Каждый из этих методов имеет свои ограничения по универсальности, дальности действия и предположениям об окружающей среде.

2.1 Подходы, основанные на пассивном зрении

Методы оценки глубины с использованием монокулярных изображений, такие как MiDaS [1] и DPT [2], демонстрируют хорошую точность на средних расстояниях (до 50 метров) благодаря использованию трансформеров, но возникают проблемы обобщения и увеличения ошибок на больших расстояниях [3]. Системы стереозрения, такие как ZED [4] и RealSense [5], предлагают более надежные оценки на средних расстояниях (до 40 метров), но их эффективность ограничена расстоянием между камерами. Попытки увеличения базового расстояния с помощью движения дронов [6] предполагают статичные сцены, а подходы на основе размеров [7] требуют заранее известных габаритов объекта.

2.2 Гибридные методы и методы с учетом рельефа местности

Чтобы устранить неоднозначность масштаба, некоторые методы объединяют визуальные данные с данными о местности. Методы Raycast [8], TRADE [9] и TanDepth [10] пересекают лучи обзора с цифровыми моделями местности или используют геопространственные оценки для локализации. Эти методы повышают точность, но опираются на данные о рельефе высокого разрешения, которые часто являются недоступными.

2.3 Использование лазеров и подвесных систем

В следующих работах лазерные дальномеры устанавливались стационарно [11] или на двухосевых подвесах [12] для дополнения визуальных данных и улучшения локализации на коротких расстояниях. Использование подвеса также играет ключевую роль в обеспечении видимости динамических целей, как это продемонстрировано в [13-14]. Однако такие системы зачастую разрабатываются исключительно для отслеживания, а не для интегрированной локализации.

2.4 Интеграция отслеживания

Надежное отслеживание объектов необходимо для поддержания визуального контакта и проведения измерений. В нашей предыдущей работе [15] мы представили гибридный трекер, сочетающий корреляционную фильтрацию и трекинг на основе трансформеров для взаимодействия БПЛА с целью. Этот трекер составляет основу процесса отслеживания объектов в нашей системе.

2.5 Резюме и вклад

Наш подход интегрирует отслеживание в реальном времени с активно управляемым лазером на подвесе и монокулярной камерой, используя данные о положении БПЛА для вычисления трехмерных координат цели. В отличие от предыдущих систем, он не зависит от известных размеров объектов, моделей рельефа или внешних карт, что делает его подходящим для разнообразных, неструктурированных сред.

3. Метод

3.1 Обзор системы

Предлагаемая система обеспечивает трехмерную локализацию объектов в реальном времени с использованием БПЛА, оснащенного монокулярной камерой, лазерным дальномером на стабилизированном подвесе и средствами бортовой обработки данных. Общая архитектура (рис. 1) состоит из нескольких взаимосвязанных модулей:

- **Модуль выбора объекта пользователем:** процесс начинается с ручного выбора цели через пользовательский интерфейс. Пользователь выбирает интересующий объект в кадре камеры.
- **Модуль отслеживания объекта:** после выбора объект непрерывно отслеживается в пространстве изображения с использованием алгоритма визуального отслеживания. Модуль выводит пиксельные координаты цели в каждом кадре.
- **Модуль отслеживания подвесом:** пиксельные координаты цели используются для управления подвесом, чтобы объект оставался в центре кадра камеры. Такое активное управление повышает надежность отслеживания и минимизирует размытие из-за движения.
- **Модуль локализации:** модуль объединяет пиксельные координаты цели, углы наклона подвеса, данные лазерного дальномера, GPS-координаты БПЛА и информацию о его ориентации для расчета трехмерного положения цели в глобальной системе координат.

Вся система функционирует на борту БПЛА в реальном времени, что обеспечивает автономную работу без внешней обработки или поддержки с земли.

3.2 Модуль отслеживания объекта

Модуль отслеживания объектов отвечает за непрерывную визуальную фиксацию выбранной цели в кадре. После первоначального выбора цели этот модуль автоматически следует за ней в последующих кадрах, оценивая ее двумерное местоположение в пиксельных координатах изображения.

Мы используем KCF-MF (Kernelized Correlation Filter-MixFormer) [15] – гибридный трекер, разработанный нашей командой. KCF-MF сочетает в себе эффективность традиционных корреляционных фильтров с низкой задержкой с точностью и устойчивостью нейронных сетей на основе трансформаторов. В нем используется механизм переключения на основе перцептивного хэширования, который при необходимости позволяет чередовать быстрые и

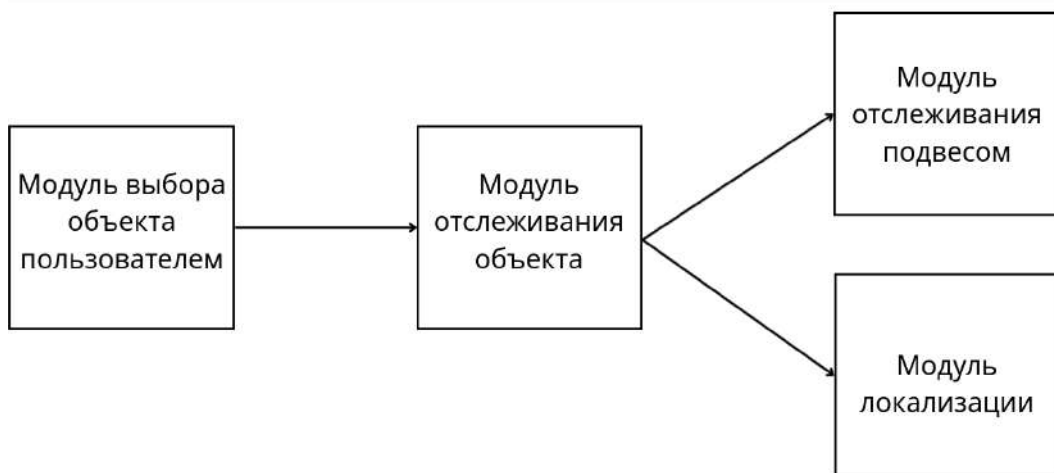


Рис. 1. Модульная архитектура предлагаемой системы локализации объектов. Процесс начинается выбора объекта пользователем, за которым следует его отслеживание, одновременно управляющее процессами контроля подвеса и локализации.

Fig. 1. Modular architecture of the proposed object localization system. The pipeline begins with user selection, followed by object tracking, which concurrently drives gimbal control and localization processes.

эффективные корреляционные фильтры с медленными, но высокоточными нейросетевыми алгоритмами. Такая гибридная конструкция обеспечивает высокую точность и низкие вычислительные затраты, что делает ее идеальной для сред с ограниченными ресурсами. На каждом временном шаге модуль выдает ограничительную рамку и центральный пиксель отслеживаемой цели. Эта информация одновременно передается в

- **Модуль отслеживания подвесом** для визуального управления;
- **Модуль локализации** для вычисления трехмерного положения цели.

3.3 Системы координат

В предложенной архитектуре используются различные системы координат для представления положения объектов и БПЛА. Точные преобразования между этими системами являются критически важными как для визуального управления, так и для локализации. Определены следующие системы координат:

- **Система камеры:** Система координат относительно оптического центра камеры. Координаты изображения из модуля отслеживания объекта выражены в этой системе.
- **Система корпуса:** Система координат, фиксированная относительно корпуса БПЛА, с началом в центре БПЛА. Ориентация и положение БПЛА описываются в этой системе.
- **Инерциальная система (NED):** Локальная система координат, фиксированная относительно Земли, с осями, направленными на Север, Восток и Вниз.
- **Глобальная система (WGS84):** Глобальная система координат, обычно связанная с GPS или другими геопривязанными данными. Окончательное вычисленное положение цели выражается в этой системе.

Более подробную информацию о преобразованиях между системами координат можно найти в работах [7] и [16].

3.4 Модуль отслеживания подвесом

Модуль отслеживания подвесом непрерывно корректирует ориентацию камеры, чтобы отслеживаемый объект оставался в центре изображения, управляя поворотами подвеса по углам рыскания и тангажа. Процесс начинается с двумерных координат изображения цели, которые преобразуются в немасштабированные трехмерные координаты в системе камеры с использованием его внутренних параметров, таких как фокусное расстояние и оптический центр. Поскольку требуется только направление на объект, а не его фактическое расстояние, масштаб полученного вектора не имеет значения. Затем этот вектор направления преобразуется в систему координат корпуса с использованием известной внешней калибровки между камерой и корпусом БПЛА. Полученный луч, выраженный в координатах тела, указывает на цель. На основе этого луча вычисляются необходимые углы рыскания и тангажа подвеса, которые передаются контроллеру подвеса для поворота камеры. Это обеспечивает постоянное визуальное выравнивание с целью в реальном времени.

Внутренние и внешние параметры камеры соответствуют стандартной модели камеры с отверстием [17].

3.5 Модуль локализации

Модуль локализации вычисляет позицию выбранной цели в глобальных координатах следующим образом:

1. Измерение дальномером: Дальномер определяет расстояние от БПЛА до цели, когда цель находится в центре кадра камеры.
2. Преобразование пикселей в систему камеры: Позиция цели выражается в масштабированных координатах камеры на основе ее положения в пикселях (из модуля отслеживания объекта) и данных дальномера.
3. Преобразование из системы камеры в систему корпуса: С использованием углов рыскания и тангажа подвеса положение цели преобразуется в систему координат корпуса БПЛА.
4. Преобразование из системы корпуса в инерциальную систему: Положение цели преобразуется в инерциальную систему с использованием ориентации БПЛА, полученной от контроллера полета.
5. GPS-координаты. Для преобразования положения цели из системы NED в GPS-координаты используется текущее положение БПЛА (широта, долгота, высота) в качестве опорной точки. Сдвиг цели в системе NED применяется к этой опорной точке с учетом азимута (направления) и расстояния по геодезической линии. В результате вычисляются новые широта, долгота и высота цели, что позволяет определить ее глобальную позицию в системе координат WGS84.

Этот процесс позволяет в реальном времени рассчитывать трехмерное положение цели в глобальных координатах, обеспечивая точную геолокацию.

4. Эксперименты

Для тестирования системы мы используем квадрокоптер со следующей конфигурацией (рис. 2):

- **Рама, моторы, регуляторы скорости (ESC):** Набор Holybro S500;
- **Полетный контроллер:** Pixhawk 6C;
- **Телеметрия:** SIYI HM30 Full HD, видеосвязь 5.8 ГГц;
- **GPS:** Holybro F9P;
- **Подвес:** Tarot 3D III Metal, 3-осевой;

- **Камера:** SIYI R1, 1080p, 30 кадров в секунду;
- **Дальномер:** Лазерный дальномер с инфракрасным лучом;
- **Аккумулятор:** Литий-полимерный аккумулятор Zeee 3S-4S;
- **Бортовой компьютер:** Jetson Orin Nano.



*Рис. 2. Holybro S500 дрон для экспериментов.
Fig. 2 Holybro S500 drone for experiments.*

Подвес был установлен на передней части дрона. Поскольку он не предназначен для одновременного размещения камеры и дальномера, была спроектирована и напечатана на 3D-принтере дополнительная крепежная деталь, обеспечивающая их размещение рядом друг с другом. Оси камеры и дальномера выровнены параллельно. Дальномер надежно закреплен и предварительно откалиброван. Расстояние между центром камеры и центром дальномера составляет примерно шесть сантиметров. На рис. 3 показан результат крепления дальномера рядом с камерой. Дальномер предоставляет не только данные о расстоянии до точки, на которую он направлен, но и угол относительно горизонта. Этот угол измеряется с большей точностью, чем угол наклона подвеса, поэтому он используется в качестве основного источника данных о наклоне.

Дальномер работает с использованием инфракрасного луча и может измерять расстояния от трех метров до 1,3 километра. Важно отметить, что поверхность должна быть твердой и не состоять из стекла или воды, так как луч должен отражаться обратно к дальномеру для точного измерения расстояния.

Система работает в реальном времени на Jetson Orin Nano. Единственным вычислительно затратным компонентом является трекер, который работает в режиме реального времени, как указано в [15]. Все остальные модули, включая управление подвесом и локализацию, являются легкими и эффективно работают параллельно.

Для тестирования системы мы разместили объект на земле и зафиксировали GPS-координаты. Затем мы запускали наш квадрокоптер на разных высотах и с разными ориентациями, пытаясь вычислить GPS-координаты при отслеживании объекта с помощью подвеса.



Рис. 3. Подвес с камерой и дальномером.
Fig. 3. Gimbal with camera and rangefinder.

4.1 Сравнение результатов

Прямое сравнение с существующей литературой представляет сложности, так как многие предыдущие работы опираются на симуляции с идеальными условиями. Напротив, наша система использует реальный подвес с лазерным дальномером на БПЛА, что вводит реалистичные ошибки, такие как вибрации, смещение датчиков и атмосферные эффекты. Кроме того, наш метод невозможно протестировать в симуляционных средах, так как он зависит от факторов и динамики реального мира, которые невозможно воспроизвести в моделируемых условиях.

Табл. 1 подводит итоги результатов. Во втором столбце указаны средние ошибки из статьи [8], основанной на экспериментах в реальном мире с использованием нескольких изображений. В третьем столбце приведены средние ошибки симуляций из работы [16], выполненных по фиксированной круговой траектории на высоте 60 метров.

Табл. 1 Абсолютные ошибки локализации для различных методов на разных расстояниях до объекта. Первый столбец содержит расстояния до объекта, а второй, третий и четвертый столбцы представляют абсолютные ошибки для метода трассировки лучей [8], нашего метода и геопространственного метода [16] соответственно.

Table 1. Absolute localization errors for various methods at different object distances. The first column lists the distances to the object, while the second, third, and fourth columns present the absolute errors for raycast [8], our method, and the geospatial [16] method, respectively.

Distance Range	Raycast [8]	Our System	geospatial [16]/simulation
40–50 m	~4.4 m	0.8m	-
50–60 m	~5.2 m	1.1m	~1.17m
60–70 m	5.62 m	1.1m	-
80–90 m	16.99 m	1.4m	-
90–100 m	-	1.6m	-
140–150 m	-	2.2m	-
300 m	-	3.3m	-

Наши эксперименты в реальных условиях показывают, что предложенная система превосходит существующие методы по точности локализации благодаря надежной оценке глубины, обеспеченной подвесом с лазерным дальномером и визуальным отслеживанием на борту.

5. Заключение

В данной работе представлена система для локализации объектов в реальном времени на основе БПЛА, объединяющая монокулярную камеру, подвес с лазерным дальномером и бортовую обработку данных. Система работает автономно в реальных условиях без необходимости в предварительных знаниях об окружающей среде или предположениях о размере объектов и рельефе местности. Полевые испытания подтвердили точность системы в локализации целей, подчеркнув ее надежность и эффективность в реальных условиях. Благодаря сочетанию активного управления подвесом и слияния данных с датчиков, система обеспечивает точную геолокацию, выполняя все вычисления на борту БПЛА в реальном режиме.

Список литературы / References

- [1]. Ranftl R., et al. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 3, 2020, pp. 1623-1637. DOI: 10.1109/TPAMI.2020.3019967.
- [2]. Ranftl R., Bochkovskiy A., Koltun V. Vision transformers for dense prediction. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 12179-12188. DOI: 10.1109/ICCV48922.2021.01196.
- [3]. Godard C., et al. Digging into self-supervised monocular depth estimation. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3828-3838. DOI: 10.1109/ICCV.2019.00393
- [4]. Stereolabs. ZED 2 Camera. Available at: <https://www.stereolabs.com/zed-2/>.
- [5]. Intel Corporation. Intel RealSense Depth Camera D455. Available at: <https://www.intelrealsense.com/depth-camera-d455/>.
- [6]. Kim I., Yow K. C. Object location estimation from a single flying camera. *UBICOMM 2015*, 2015, p. 95.
- [7]. Sanyal S., Bhushan S., Sivayazi K. Detection and location estimation of object in unmanned aerial vehicle using single camera and GPS. *2020 First International Conference on Power, Control and Computing Technologies (ICPC2T)*, IEEE, 2020, pp. 73-78. DOI: 10.1109/ICPC2T48082.2020.9071439.
- [8]. Paulin, et al. Application of the raycast method for person geolocalization and distance determination using UAV images in real-world land search and rescue scenarios. *Expert Systems with Applications*. DOI: 10.1016/j.eswa.2023.121495.
- [9]. Proença P. F., et al. Trade: Object tracking with 3D trajectory and ground depth estimates for UAVs. *2023 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2023, pp. 3325-3331. DOI: 10.1109/ICRA48891.2023.10161192.
- [10]. Florea H., Nedevschi S. TanDepth: Leveraging Global DEMs for Metric Monocular Depth Estimation in UAVs. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 2025. DOI: 10.1109/JSTARS.2025.3531984.
- [11]. Yates D. J. Monocular Vision Localization Using a Gimbaled Laser Range Sensor. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2010, 105 pp.
- [12]. Ge H., et al. An object localization system using monocular camera and two-axis-controlled laser ranging sensor for mobile robots. *IEEE Access*, vol. 9, 2021, pp. 79214-79224. DOI: 10.1109/ACCESS.2021.3084153.
- [13]. Cunha R., et al. Gimbal control for vision-based target tracking. *Proceedings of the European Conference on Signal Processing*, 2019.
- [14]. Regner D. J., et al. Object tracking control using a gimbal mechanism. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 43, 2021, pp. 189-19. DOI: 10.5194/isprs-archives-XLIII-B1-2021-189-2021.
- [15]. Sardaryan A., et al. An accurate real-time object tracking method for resource-constrained devices. *Proceedings of the Institute for System Programming of the RAS*, vol. 36, no. 3, 2024, pp. 283-294. DOI: 10.15514/ISPRAS-2024-36(3)-20.
- [16]. Pai P., Naidu V. P. S. Target geo-localization based on camera vision simulation of UAV. *Journal of Optics*, vol. 46, 2017, pp. 425-435. DOI: 10.1007/s12596-017-0395-0.
- [17]. Hartley R., Zisserman A. *Multiple View Geometry in Computer Vision*. Cambridge University Press, Cambridge, 2004, 655 pp.

Информация об авторах / Information about authors

Вардан СААКЯН – научный сотрудник Центра передовых программных технологий (CAST) и аспирант Российско-Армянского университета, специализируется на математическом и программном обеспечении вычислительных систем. Получил степень бакалавра по информатике и прикладной математике в Национальном политехническом университете Армении (2021) и степень магистра по интеллектуальным системам и робототехнике в Российско-Армянском университете (2023). Его исследования посвящены беспилотным летательным аппаратам, компьютерному зрению и обучению с подкреплением.

Vardan SAHAKYAN is a researcher at the Center of Advanced Software Technologies (CAST) and a postgraduate student at Russian-Armenian University, specializing in mathematical and software support for computing systems. He holds his B.Sci. in informatics and applied mathematics from the National Polytechnic University of Armenia (2021) and an M.Sc. in intellectual systems and robotics from Russian-Armenian University (2023). His research focuses on UAVs, computer vision, and reinforcement learning.

Ваагн МЕЛКОНЯН получил степень бакалавра в области информатики и прикладной математики в Национальном Политехническом Университете Армении, Армения, в 2021 году. В 2023 году он получил степень магистра в области интеллектуальных систем и робототехники в Российско-Армянском Университете, Армения. В настоящее время он занимается аспирантурой по математическому и программному обеспечению вычислительных машин, комплексов и компьютерных сетей в Российско-Армянском Университете, Армения. Он также является исследователем в Центре Передовых Программных Технологий (CAST). Его исследовательские интересы включают БПЛА, компьютерное зрение и алгоритмы управления.

Vahagn MELKONYAN received his B.Sci. in Informatics and Applied Mathematics from the National Polytechnic University of Armenia, Armenia, in 2021. In 2023, he earned his M.Sc. degree in Intellectual Systems and Robotics from Russian-Armenian University, Armenia. He is currently pursuing a Ph.D. in Mathematical and Software Support for Computing Machines, Complexes, and Computer Networks at Russian-Armenian University, Armenia. He is also a researcher at the Center of Advanced Software Technologies (CAST). His research interests include UAVs, computer vision, and control algorithms.

Севак САРГСЯН получил степени бакалавра и магистра в области информатики и прикладной математики в Ереванском Государственном Университете, Армения, в 2010 и 2012 годах соответственно. Позже, в 2016 году, он получил степень кандидата физ.-мат. наук в области математического и программного обеспечения вычислительных машин, комплексов и компьютерных сетей в Институте системного программирования имени Иваницова Российской академии наук. В настоящее время он является заведующим кафедрой Системного Программирования в Российско-Армянском Университете, Армения. Его исследовательские интересы включают технологии компиляторов, безопасность программного обеспечения и тестирование программного обеспечения.

Sevak SARGSYAN received his B. Sci. and M. Sci. degrees in informatics and applied mathematics from Yerevan State University, Armenia, in 2010 and 2012, respectively. He later in 2016, obtained his Cand. Sci. (Phys.-Math.) degree in mathematical and software support for computing machines, complexes, and computer networks from the Ivannikov Institute for System Programming of the Russian Academy of Sciences. Presently, he serves as the head of the system programming department at Russian-Armenian University, Armenia. His research interests include compiler technologies, software security, and software testing.



DOI: 10.15514/ISPRAS-2025-37(4)-12

Именная морфология первого печатного памятника тверской карельской письменности

И.П. Новак, ORCID: 0000-0002-9436-9460 <bel.irina@rambler.ru>

*Институт языка, литературы и истории Карельского научного центра РАН,
Россия, 185910, Республика Карелия, г. Петрозаводск, ул. Пушкинская, д. 11.*

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

Аннотация. В статье описана именная словоизменительная система языка брошюры 1804 г. «Переводъ нѣкоторыхъ молитвъ и сокращеннаго катихизиса на Корельскій языкъ» – первого печатного памятника тверской карельской письменности. Актуальность обусловлена отсутствием систематического лингвистического анализа источника, содержащего уникальный корпус данных по исторической грамматике и диалектологии карельского языка. Объем карельского текста составляет порядка 2,5 тысяч лексем, из которых более половины представлены именными частями речи. В работе подвергнуты анализу все грамматические категории имени: число, падеж, притяжательность. Исследование проведено с использованием инструментов лингвистической платформы LingvoDoc. Материалы источника сопоставлены с новописьменным вариантом тверского карельского языка, а также с данными по тверским говорам середины XX в. из диалектной базы Murreh.

Именная словоизменительная система перевода не демонстрирует существенных отличий от толмачевской группы говоров южного диалекта собственно карельского наречия, что подтверждает сделанный ранее вывод о диалектной принадлежности источника. В ходе исследования установлено, что язык памятника сохранил архаичные черты, утратившиеся в современных тверских карельских говорах. К числу таких архаизмов относятся: использование показателя множественного числа *-й* с многосложными одноосновными именами на дифтонг, окончания множественного числа генитива *-ень*, множественного числа партитива *-а*, иллатива *-же*, адессива-аллатива *-ла / -ла*, инессива *-са / -са*, абессива *-та*, притяжательные суффиксы, а также употребление послеложных конструкций для передачи значений комитатива и аппроксиматива. Инновационным является распространение форманта множественности *-лой- / -лйой-* на одноосновные имена с гласной основой на *-е*, а также тенденция перехода от посессивных суффиксов к генитивным конструкциям.

Ключевые слова: карельский язык; тверские карелы; памятник письменности; диалектология; история языка; морфология; именное словоизменение; лингвистическая платформа LingvoDoc.

Для цитирования: Новак И.П. Именная морфология первого печатного памятника тверской карельской письменности. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 199–214. DOI: 10.15514/ISPRAS-2025-37(4)-12.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 25-78-20006, <https://rscf.ru/project/25-78-20006/>. Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П.Иванникова РАН – ЦКП ИСП РАН.

Nominal Morphology of the Earliest Printed Piece of Tver Karelian Written Language Heritage

I.P. Novak, ORCID: 0000-0002-9436-9460 <bel.irina@rambler.ru>

*Institute of Linguistics, Literature and History of KarRC RAS,
11, Pushkinskaya st., Petrozavodsk, 185910, Russia.*

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

Abstract. The article describes the nominal inflexion systems of the language used in the pamphlet dated 1804 “Translation of some prayers and the shorter catechism in the Karelian language” – the first printed piece of the Tver Karelian written language heritage. This study was necessitated by the lack of a previous systematic linguistic analysis of this source, which contains a unique corpus of data on the Karelian historical grammar and dialectology. The Karelian text is made of around 2,500 lexemes, more than a half of them represented by nominal parts of speech. The analysis made in this article covers all grammatical categories of nominals: number, case, possessiveness. The study was conducted using the tools of the LingvoDoc linguistic platform. The material found in this source was compared against the new-script variant of the Tver Karelian language and with data on mid-20th century Tver Karelian dialectal variants from the Murreh dialectal database. The nominal inflexion system in the translated text has no substantial differences from the Tolmachyovo sub-dialect group of the southern dialect of Karelian Proper, supporting the dialectal attribution of the source made previously. One of the findings of this study is that the language of this source preserves some archaic traits that had been lost from modern Tver Karelian sub-dialects. Such archaisms include the use of the plurality indicator *-ŭ-* with polysyllabic single-stem nominals ending in a diphthong, endings *-ень* in the genitive plural, *-а* in the partitive plural, *-же* in the illative, *-ла / -лаа* in the adessive-allative, *-са / -саа* in the inessive, *-та* in the abessive, possessive suffixes, and the use of postpositional constructs to render the commitative and approximative meanings. A novel feature is the expansion of the plurality formant *-лой-* / *-лой-* to single-stem nominals with an *-e* vowel stem and a trend to abandon possessive suffixes for a wider use of genitive constructs.

Keywords: Karelian; Tver Karelians; written language heritage; dialectology; history of language; morphology; nominal inflexion; linguistic platform LingvoDoc.

For citation: Novak I.P. Nominal morphology of the earliest printed piece of Tver Karelian written language heritage. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 199-214 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-12.

Acknowledgements. The reported study was granted by Russian Science Foundation, project no. 25-78-20006, <https://rscf.ru/project/25-78-20006/>. The results were obtained using the services of the Center for Collective Use of the Institute for System Programming named after V.P. Ivannikov of the Russian Academy of Sciences – CCU ISP RAS.

1. Введение

В январе 1803 г. Священный Синод разослал по епархиям, «в которых имелись инородцы», указы о необходимости перевода на их языки катехизиса и церковных молитв [1]. В 1804 г. свет увидели переводы на «олонецкий» (ливиковское наречие карельского языка) и «корельский» (собственно карельское наречие) языки. Графо-фонетический и фонологический анализ «корельского» источника [2] позволяет относить его диалектную основу к толмачевской группе говоров южного диалекта собственно карельского наречия, то есть к говорам карельского языка центральных районов современной Тверской области. Памятник объемом в 2,5 тыс. лексем представляет собой первый известный переводной текст, пригодный для анализа становления морфологии карельских говоров Верхневолжья, поскольку словарные записи XVII–XVIII вв. [3, 4] в этом отношении не показательны.

«Переводъ нѣкоторыхъ молитвъ и сокращеннаго катихизиса на Корельскій языкъ» был выполнен и опубликован, когда карелы проживали на территории Верхневолжья около двух

столетий. В языке того периода еще сохранялись отдельные архаичные древнекарельские черты, но, с другой стороны, уже наметились собственные диалектные особенности. При этом тверские говоры начала XIX в., являясь основным средством общения в регионе, еще не были подвержены сильному влиянию со стороны русского языка [5].

В статье представлен анализ именной словоизменительной системы языка памятника (763 имени существительных, 177 имен прилагательных, 418 местоимений и 24 числительных). Описаны все грамматические категории: падеж, число, притяжательность. Сравнение в современной финнистике принято относить к словообразованию [6].

В карельском языке образование словоформ происходит путем последовательного присоединения содержащих грамматическое значение аффиксов к основе слова. Для именной словоизменения характерна формула: лексическая основа + (показатель числа) + (падежное окончание) + (притяжательный суффикс). В одной форме может присутствовать только один словоизменительный показатель конкретной грамматической категории.

Исследование проведено с использованием инструментов LingvoDoc [7]. Аннотированный текст [8], конкорданс [9] и морфологический словарь [10] перевода представлены на сайте лингвистической платформы. Данные источника сопоставлены с грамматикой новописьменного варианта тверского карельского языка, основанного на базе толмачевских говоров [11], а также с материалами по именной морфологии тверских карельских говоров из диалектной базы карельского языка Murteh [12]. Речь идет об оцифрованных и закодированных ответах на более 200 вопросов по именной морфологии из «Программ по собиранию материалов для диалектологического атласа карельского языка», заполненных в полевых условиях в 1957–1967 гг. в 60 населенных пунктах Калининской области [13].

2. Грамматическая категория числа

Карельскому языку свойственна бинарная структура категории числа. Соотношение единственного и множественного числа в языке памятника составляет 82% к 18%. Единственное число не имеет специального показателя, тогда как множественное обладает комплексом формантов, употребление которых зависит от словоизменительного типа имени и от конкретной падежной формы.

Для образования **множественного числа аккузатива и номинатива** используется окончание **-тъ**:

- PL.NOM: *azie-тъ* ‘дело’¹, *kirъa-тъ* ‘книга’, *kirъютукше-тъ* ‘писание’, *kiza-тъ* ‘игра’, *liika-тъ* ‘икона’, *nađio-тъ* ‘лицо’, *пайва-тъ* ‘день’, *virre-тъ* ‘песня’, *руаво-тъ* ‘работа’, *ванга-тъ* ‘старый’, *ендизе-тъ* ‘прежний’, *кевга-тъ* ‘бедный’, *пага-тъ* ‘плохой’, *нюга-тъ* ‘священный’, *шүүре-тъ* ‘великий’ (ср. литер. *azie-t*, *kirja-t*, *kirjutukse-t*, *kiza-t*, *liika-t*, *nävö-t*, *päivä-t*, *virre-t*, *ruavo-t*, *vanha-t*, *endize-t*, *keyhä-t*, *paha-t*, *pyhä-t*, *šuuere-t*);
- PL.ACC: *azie-тъ* ‘дело’, *міеле-тъ* ‘мысль’, *рїага-тъ* ‘грех’, *тахтомизе-тъ* ‘намерение’, *руаво-тъ* ‘работа’, *велла-тъ* ‘долг’, *иччіе-тъ* ‘сам’, *ома-тъ* ‘свой’, *валлагизе-тъ* ‘свободный’, *күмбазе-тъ* ‘который’, *нюга-тъ* ‘священный’ (ср. литер. *azie-t*, *miele-t*, *riähä-t*, *tahtomize-t*, *ruavo-t*, *vella-t*, *iččie-t*, *oma-t*, *vällähize-t*, *kumbaze-t*, *pyhä-t*).

Для образования форм множественного числа косвенных падежей используются суффиксы *-и* / *-ы* / *-і* / *-й* / *-ий*-, *-лой* / *-лій* / *-лоі*-, *-ло* / *-ліо* / *-ліо*-.

Дистрибуция алломорфов наиболее древнего показателя множественности **-и** / **-ы** / **-і** / **-й** / **-ий** (*-i) в тексте памятника представлена следующим образом:

¹ Перевод приводится в начальной форме слова.

- 1) **-и- / -ы-** выступает перед падежными показателями, начинающимися на согласный, используется с одно- и двусловными именами с гласной основой на *а-, а-, е-*, при этом происходит выпадение гласного при присоединении аффикса множественности: *гюв-и-тта* ‘хороший-PL-AB’, *кевг-и-лла* ‘бедный-PL-ALL’, *шилм-и-сса* ‘глаз-PL-IN’, *рїаг-и-ста* ‘грех-PL-ELA’, *пюг-и-ень* ‘священный-PL-GEN’, *пұхтаг-ы-нь* ‘чистый-PL-GEN’, *юкс-и-сса* ‘один-PL-IN’, *мїютюз-и-лла* ‘какой-PL-AD’, *ализ-и-сса* ‘нижний-PL-IN’, *енжимаз-и-кси* ‘первый-PL-TRANS’, *найз-и-ста* ‘жена-PL-ELA’ (ср. литер. *huv-i-ttä* < *huvä*, *keyh-i-llä* < *keyhä*, *šilm-i-ssä* < *šilmä*, *riäh-i-stä* < *riähkä*, *pyh-i-n* < *pyhä*, *puhtah-i-n* < *puhtaha-*, *yks-i-ssä* < *yhte-*, *miittyz-i-llä* < *miittyze-*, *aliz-i-ssa* < *alize-*, *enžimäz-i-ksi* < *enžimäze-*, *naiz-i-sta* < *naize-*);
- 2) **-ї-** используется перед падежными показателями, начинающимися на гласный. Он употребляется:
 - с одно- и двусловными именами с гласной основой на *а-, а-, е-*, при этом гласный основы бесследно исчезает: *рїахк-ї-ень* / *рїагк-ї-ень* / *рїаг-ї-ень* ‘грех-PL-GEN’, *рїагк-ї-е* / *рїахк-ї-е* ‘грех-PL-PRT’, *пюг-ї-ень* ‘священный-PL-GEN’, *пюг-ї-е* ‘священный-PL-PRT’, *шилм-ї-ень* ‘глаз-PL-GEN’, *иг-ї-ень* ‘век-PL-GEN’, *мїег-ї-ень* ‘муж-PL-GEN’, *рїахкагиз-ї-ень* ‘грешный-PL-GEN’, *рїагкагиз-ї-е* ‘грешный-PL-PRT’, *инегмиз-ї-ень* ‘человек-PL-GEN’, *тойз-ї-ень* ‘другой-PL-GEN’, *тойз-ї-е* ‘другой-PL-PRT’, *елав-ї-е* ‘живой-PL-PRT’, *глад-ї-е* ‘нужда-PL-PRT’, *вїераг-ї-е* ‘чужой-PL-PRT’, *пұхтаг-ї-е* ‘чистый-PL-PRT’, *ом-ї-е* ‘свой-PL-PRT’, *вангемб-ї-е* ‘старший-PL-PRT’, *гунуки-ї-е* ‘строение-PL-PRT’, *войматуки-ї-е* ‘слабость-PL-PRT’, *кодилаз-ї-е* ‘домашний-PL-PRT’, *кодигиз-ї-е* ‘домочадец-PL-PRT’, *мїютюз-ї-е* ‘какой-PL-PRT’, *мїелегиз-ї-е* ‘милый-PL-PRT’, *колм-ї-е* ‘три-PL-PRT’, но *свато-ї-ень* ‘святой-PL-GEN’ (ср. литер. *riäh-i-stä*, *riähk-i-n*, *riähk-i-e* < *riähkä*, *pyh-i-n*, *pyh-i-e* < *pyhä*, *šilm-i-n* < *šilmä*, *ig-i-n* < *igä*, *mieh-i-n* < *miehe-*, *riähkähiz-i-n*, *riähkähiz-i-e* < *riähkähize-*, *inehmiz-i-n* < *inehmize-*, *toiz-i-n*, *toiz-i-e* < *toize-*, *eläv-i-e* < *elävä*, *häd-i-e* < *hädä*, *vierah-i-e* < *vieraha-*, *puhtah-i-e* < *puhtaha-*, *om-i-e* < *oma*, *vanhemb-i-e* < *vanhembra-*, *huonukš-i-e* < *huonukše-*, *voimatukš-i-e* < *voimatukše-*, *kodilaz-i-e* < *kodilaze-*, *kodihiz-i-e* < *kodihize-*, *miittyz-i-e* < *miittyze-*, *mielehiz-i-e* < *mielehize-*, *kolmi-loi-da*¹ < *kolme-*, *sv'atoi-loi-n* < *sv'atoi*);
 - с односложными именами с гласной основой на *-а*, переходящим в *-о* в позиции перед показателем множественности: *апосто-ї-ень* ‘апостол-PL-GEN’, *кашкюлїо-ї-ень* ‘слуга-PL-GEN’, *пророко-ї-ень* / *пророкко-ї-ень* ‘пророк-PL-GEN’, *ристиканжо-ї-ень* ‘христианин-PL-GEN’ (ср. литер. *apostolo-i-n* < *apostola*, *käškylö-i-n* < *käškylä*, *prorokko-i-n* < *prorokka*, *ristikanžo-i-n* < *ristikanža*). К последней группе можно также отнести случаи, в которых показатель множественности входит в йотированный гласный окончания, что не позволяет выделить его в структуре словоформы. На его присутствие указывает переход гласного *-а* основы в *-о*: *геро-ї-ень* ‘господин-PL-GEN’, *герро-ї-а* ‘господин-PL-PRT’, *абивойндо-ї-а* ‘обида-PL-PRT’, *авко-ї-а* ‘явка-PL-PRT’, *юмало-ї-а* ‘Бог-PL-PRT’, *лїйко-ї-а* ‘икона-PL-PRT’, *пайко-ї-а* ‘место-PL-PRT’, *шано-ї-а* ‘слово-PL-PRT’ (ср. литер. *herro-i-da* < *herra*, *abivoindo-i-da* < *abivoinda-*, *javko-i-da* < *javka*, *jumalo-i-da* < *jumala*, *liiko-i-da* < *liika*, *paikko-i-da* < *paikka*, *šano-i-da* < *šana*);
- 3) **-й-** употребляется в позиции после гласного основы, образуя с этим гласным сужающийся дифтонг. Он выступает перед падежными показателями, начинающимися на согласный. Формант используется:
 - с двусложными и многосложными односложными именами с гласной основой на

¹ Здесь и далее в перечне соответствий из литературного языка подчеркнуты несовпадения в образовании анализируемой формы.

-а и многосложными именами с основой на -а, переходящими в -о / -iо / -iо перед показателем множественности: *пaгo-й-ла* 'плохой-PL-AD', *пaгo-й-са* 'плохой-PL-IN', *гeppo-й-ла* 'господин-PL-AD', *шaнo-й-ла* 'слово-PL-ALL', *кирвiо-й-са* 'книга-PL-IN', *присaгo-й-са* 'присяга-PL-IN', *пpуазднiеко-й-на* 'праздник-PL-ESS', *кашiекеннiо-й-ла* 'заповедь-PL-AD', *кашiеннiо-й-са* 'заповедь-PL-IN' (ср. литер. *paho-i-lla*, *paho-i-ssa* < *paha*, *herro-i-lla* < *herra*, *šano-i-lla* < *šana*, *kirjo-i-ssa* < *kirja*, *pris'ago-i-ssa* < *pris'aga*, *pruazniekko-i-na* < *pruazniekka*, *käššennö-i-llä*, *käššennö-i-ssä* < *käššendä*);

- с одноосновными именами с гласной основой на расширяющийся дифтонг, при этом первый гласный компонент дифтонга выпадает перед показателем множественности: *азе-й-та* 'дело-PL-AB', *азе-й-ла* 'дело-PL-ALL', *азе-й-са* 'дело-PL-IN', *кyоле-й-да* 'покойник-PL-PRT' (ср. литер. *azie-loi-tta*, *azie-loi-lla*, *azie-loi-ssa* < *azie*, *kuolie-loi-da* < *kuolie*);

- 4) **-iй-**, содержащий удвоенный формант множественного числа, представлен в единственном примере: *ом-й-са* 'свой-PL-IN' (ср. литер. *om-i-ssa* < *oma*). Явление плеоназма аффикса – типичная инновационная черта, характеризующая северо-восточные толмачевские и валдайские (ID 1182, 2016 [14]) говоры собственно карельского наречия карельского языка.

Показатель множественности **-лой-** / **-лiой-** / **-лой-**, **-ло-** / **-лiо-** / **-лiо-**, восходящий к древнекарельскому языку [15-16], используется:

- 1) с одноосновными именами с основой на краткие лабиализованные гласные и гласный -и, при этом в позиции перед йотированным гласным показателя возможно слияние с ним конечного -й аффикса: *pyado-лой-ла* 'работа-PL-AD', *myatto-лой-ла* 'отец-PL-AD', *myatto-лой-нъ* 'отец-PL-GEN', *myatto-лоi-а* / *myatto-ло-а* 'отец-PL-PRT', *myatto-ло-ень* 'отец-PL-GEN', *malittu-лой-са* 'молитва-PL-IN', *myamo-лой-нъ* 'мать-PL-GEN', *myamo-ло-а* 'мать-PL-PRT', *izrastu-лой-ла* 'подобие-PL-ALL' (ср. литер. *ruado-loi-lla*, *tuatto-loi-lla*, *tuatto-loi-n*, *tuatto-loi-da*, *malittu-loi-ssa*, *muamo-loi-n*, *muamo-loi-da*, *izrastu-loi-lla*);
- 2) с одноосновными именами с основой на -е, переходящим в -и: *генги-лiо-ень* 'душа-PL-GEN' (ср. литер. *hengi-löi-n* < *henge-*);
- 3) с двусловными двусложными именами с гласной основой на -е, переходящим в -и: *мiели-лiой-са* 'мысль-PL-IN', *мiели-лiой-ста* 'мысль-PL-ELA', *мiеги-лiо-а* 'муж-PL-PRT' (ср. литер. *miel-i-ssä*, *miel-i-stä* < *miele-*, *mieh-i-e* < *miehe-*).

Отсутствие унификации в образовании словоизменительных основ множественного числа в тексте перевода демонстрируют имена с основами на -е, как одноосновные (*колм-й-е* 'три-PL-PRT', но *генги-лiо-ень* 'душа-PL-GEN'), так и двусложные двусловные (*юкс-и-сса* 'один-PL-IN', *мiег-й-ень* 'муж-PL-GEN', но *мiеги-лiо-а* 'муж-PL-PRT', *мiели-лiой-ста* 'мысль-PL-ELA'). В диалектной речи середины XX в. с одноосновными именами используется показатель **-лоi-** / **-лiоi-** (ID 2283, 2288 [14]), а с двусловными – преимущественно **-i-** (ID 2282 [14]), что нашло отражение в новописьменном варианте языка.

Использование форманта множественности **-й-** с одноосновными именами на дифтонг, отраженное в тексте анализируемого источника (*азе-й-та* 'дело-PL-AB', *азе-й-ла* 'дело-PL-ALL', *азе-й-са* 'дело-PL-IN'), совпадает с диалектным представительством явления в большей части тверских карельских говоров (ID 2284, 2285, 2289 [14]). Отмеченная в диалектных данных тенденция к унификации в пользу употребления показателя **-лоi-** / **-лiоi-** (аналогично односложным именам на дифтонг) впоследствии нашла отражение в нормированном варианте языка.

Представленная в тексте перевода система распределения показателей множественного числа косвенных падежей указывает на ее незавершенный характер в древнекарельский период. Изначально в праязыке использование форманта **-лоi-** / **-лiоi-** было ограничено

одноосновными именами с основой на краткий лабиализованный гласный или гласный *-i* [17]. Отмеченное в памятнике письменности распространение данного показателя на имена с основой на *-e*, переходящим в *-и*, можно отнести к инновационным явлениям. В свою очередь, употребление форманта *-й* с многосложными одноосновными именами на дифтонг, в сравнении с современным представительством категории, относится к архаизмам.

3. Падежная система

Падежная система памятника представлена 13 падежами (табл. 1).

Табл. 1. Частота встречаемости падежных форм в тексте памятника и падежные окончания.

Table 1. Frequency of occurrence of case forms in the text of the monument and case endings.

Падеж	Число фиксаций			Падежные показатели	
	всего	ед. ч.	мн. ч.	ед. ч.	мн. ч.
Номинатив	391	368	23	—	-тъ
Аккузатив	54	35	19	—, -нъ	-тъ
Генитив	275	192	83	-нъ	-нъ, -енъ
Партитив	306	240	66	-а / -а, -е, -о, -да / -да, -та / -та	-е, -да, -а
Эссив	19	18	1	-на / -на	
Транслатив	8	7	1	-кии, -кси	-кси
Инессив	75	60	15	-иша / -иша, -сса, -са / -са	-сса, -са / -са
Элатив	43	33	10	-шта / -шта, -ста / -ста	-ста / -ста
Иллатив	70	69	1	-хъ, -же	
Адессив	58	51	7	-лла / -лла, -ла / -ла	
Аблатив	4	4	0	-лда / -лда	
Аллатив	75	56	19	-лла / -лла, -ла / -ла	
Абессив	4	1	3	-тта, -та	

К грамматическим падежам относятся номинатив, аккузатив, генитив и партитив. Единственное число **номинатива** (исходной формы) не имеет падежного окончания.

Показателем **генитива единственного числа** является **-нъ**, присоединяющийся к слабой гласной основе слова, напр., *генге-нъ* ‘душа’, *инегмизе-нъ* ‘человек’, *кїа-нъ* ‘рука’, *міеле-нъ* ‘мысль’, *пойа-нъ* ‘сын’, *русси-нъ* / *ристѣ-нъ* ‘крест’, *тове-нъ* ‘правда’, *туато-нъ* ‘отец’, *вее-нъ* ‘вода’, *віа-нъ* ‘сила’, *иа-нъ* ‘век’, *тайвага-нъ* ‘небо’, *руаво-нъ* ‘работа’, *папи-нъ* ‘священник’, *кирикїо-нъ* ‘церковь’, *луїа-нъ* ‘твердый’, *магіе-нъ* ‘сладкий’, *елава-нъ* ‘живой’ (ср. литер. *henge-n*, *inehmize-n*, *kiä-n* < *käde*-, *miele-n*, *poija-n* < *poiga*, *rissi-n* < *risti*, *tove-n* < *tode*-, *tuato-n* < *tuatto*, *veje-n* < *vede*-, *viä-n* < *väge*-, *ijä-n* < *igä*, *taivaha-n*, *ruavo-n* < *ruado*, *papi-n* < *rappi*, *kirikö-n* < *kirikkö*, *luja-n*, *magie-n*, *elävä-n*).

Формы **генитива множественного числа** складываются следующим образом:

- 1) со словоизменительными основами множественного числа на *-і-*, *-и-* используется окончание генитива **-енъ**, напр., *пюгі-енъ* / *пюги-енъ* ‘священный’, *ріахкі-енъ* / *ріагкі-енъ* / *ріагі-енъ* ‘грех’, *шилмі-енъ* ‘глаз’, *молебї-енъ* ‘оба’, *міегї-енъ* ‘муж’, *кумбазї-енъ* ‘который’, *тоїзі-енъ* ‘другой’, *апостолої-енъ* / но *апостолой-нъ* ‘апостол’, *пророкої-енъ* / *пророккої-енъ* ‘пророк’, *ристиканжої-енъ* / но *ристиканжо-енъ* ‘христианин’ (ср. литер. *pyhi-n*, *riähki-n*, *šilmi-n*, *molembi-n*, *miehi-n*, *kumbazi-n*, *toizi-n*, *apostoloi-n*, *prorokkoi-n*, *ristikanžoi-n*);
- 2) со словоизменительными основами множественного числа на *-ло-* / *-ліо-* возможно употребление как окончания генитива множественного числа **-енъ**, так и генитива

единственного числа **-нъ**, напр., *мѹамолой-нъ* ‘мать’, *мѹатмолой-нъ* ‘отец’, *мѹатмоло-ень* ‘отец’, *генгиліо-ень* ‘душа’ (ср. литер. *tuamo-loi-n*, *tuatto-loi-n*, *hengi-löi-n*).

В современных тверских карельских говорах (ID 2294 [14]) и в новописьменном варианте языка используется единое окончание генитива **-п** (=нъ), при этом в литературе по карельской диалектологии отмечается возможность употребления форм на **-еп** (=ень) в толмачевских говорах начала XX в. [18]. Упрощение архаичной системы генитивных форм множественного числа относится к первой половине XX в.

Падеж тотального объекта **аккузатив** в карельском языке не обладает собственным окончанием. В единственном числе его функции распределены между генитивом и номинативом, напр., ... *кумбане мѹлма* и *кайкенъ мѹаилманъ лѹади* ... ‘... который меня и весь свет создал ...’, ... *кайкенъ гибенъ мѹанъ паганюдѹа* ... ‘... и все мое тело загрязнил ...’, ... *шѹага үүжси ваги* и *юмаланъ абү* ... ‘... получить новую силу и благодать ...’, *Мѹйсса шѹоватта гювиттѹа* ... ‘Помни день субботний, чтобы святить ...’, во множественном – он совпадает с формой номинатива множественного числа, напр., ... *аттѹа кайки омаѹ рѹавотъ* ... ‘... оставить все свои работы ...’, ... *тѹедаѹ иччѹетъ* и *мѹанъ мѹелетъ* и *тахтомизетъ* ... ‘... знает самые мысли наши и намерения ...’.

Для образования форм **партитива единственного числа** в источнике в зависимости от словоизменительного типа имен и огласовки ауслота используются окончания **-а / -а, -о, -е, -да / -да, -та / -та**. Их распределение не отличается от современного представительства:

- 1) с одноосновными именами с кратким гласным **-а / -а** на конце основы используются показатели партитива **-а / -а**, присоединение которых приводит к образованию на конце словоформы расширяющихся дифтонгов *ѹа, ѹа, ѹа, ѹа*, напр., *карьѹ-а* ‘стадо’, *палкѹ-а* ‘плата’, *рагѹ-а* ‘деньги’, *табѹ-а* ‘обычай’, *пагѹ-а* ‘плохой’, *гѹргѹ-а* ‘бык’, *кюлѹ-а* ‘деревня’, *пайѹ-а* ‘день’, *лейѹ-а / лейѹ-а* ‘хлеб’, *тѹгѹ-а / тѹгѹ-а* ‘пустое’, *игѹ-а* ‘век’, *гюѹ-а* ‘добро’, *тѹгѹ-а* ‘священный’ (ср. литер. *karju-a* < *karja*, *palku-a* < *palka*, *rahu-a* < *raha*, *tabu-a* < *taba*, *pahu-a* < *paha*, *härgi-ä* < *härkä*, *kyli-ä* < *kylä*, *päivi-ä* < *päivä*, *tyhji-ä* < *tyhjä*, *igi-ä* < *igä*, *hyvi-ä* < *hyvä*, *pyhi-ä* < *pyhä*);
- 2) с одноосновными именами с кратким гласным **-о** на конце основы используется показатель партитива **-о**, присоединение которого приводит к образованию дифтонга *ѹо*, напр., *мѹамѹ-о* ‘мать’, *палѹ-о* ‘пожар’, *вѹерѹ-о* ‘вера’, *шѹвиндѹ-о* ‘согласие’ (ср. литер. *tuatu-o* < *tuamo*, *palu-o* < *palo*, *vieru-o* < *viero*, *šovindu-o* < *šovindo*);
- 3) с одноосновными именами с кратким гласным **-ү** на конце основы используется показатель партитива **-о**, присоединение которого также приводит к образованию дифтонга *ѹо*, напр., *сүүдѹ-о* ‘суд’, *дүүхѹ-о* ‘дух’, *абѹ-о* ‘помощь’ (ср. литер. *suudu-o* < *suudu*, *duuhu-o* < *duuhu*, *abu-o* < *abu*);
- 4) с одноосновными именами с кратким гласным **-е** на конце основы используется показатель партитива **-е**, присоединение которого приводит к образованию дифтонга *ѹе*, напр., *генгѹ-е* ‘сердце’, *нүрмѹ-е* ‘луг’, *нимѹ-е* ‘имя’ (ср. литер. *hengi-e* < *henge*, *nurmi-e* < *nurme*, *nimi-e* < *nime*);
- 5) с одноосновными именами с кратким гласным **-и** на конце основы используется показатель партитива **-е**, присоединение которого также приводит к образованию дифтонга *ѹе*, напр., *израстѹ-е* ‘подобие’, *кодѹ-е* ‘дом’, *недѹлѹ-е* ‘неделя’ (ср. литер. *izrasti-e* < *izrasti*, *kodi-e* < *kodi*, *nedeli-e* < *nedeli*);
- 6) с одноосновными именами на дифтонг используется показатель партитива **-да / -да**, напр., *мѹа-да* ‘земля’, *азѹе-да* ‘дело’, *сѹатой-да* ‘святой’, *нѹа-да* ‘конец’, *гибѹе-да* ‘тело’ (ср. литер. *tua-da*, *azie-da*, *svatoi-da*, *piä-dä*, *hibie-dä*);
- 7) с двуосновными именами с согласной основой на звонкий согласный используется показатель партитива **-да / -да**, напр., *мѹел-да* ‘мысль’ (ср. литер. *miel-dä*);
- 8) с двуосновными именами с согласной основой на глухой согласный используется

показатель партитива *-та / -та*, напр., *ках-та* ‘два’, *параиш-та* ‘лучший’, *вѣраиш-та* ‘чужой’, *валегуш-та* ‘ложь’, *тойс-та* ‘другой’, *найс-та* ‘жена’, *лайсүт-та* ‘лень’, *юх-та* ‘один’, *тервегуш-та* ‘здоровье’, *изагис-та* ‘вечный’, *инегмис-та* ‘человек’, *ютиот-та* ‘подобный’, *мийтиот-та* ‘какой’, *елавиот-та* ‘воскрешение’, *айагис-та* ‘временный’, *гевос-та* ‘лошадь’, *куммас-та* ‘который’, *наголас-та* ‘беглый’, *говкас-та* ‘глупый’ (ср. литер. *kah-ta*, *paraš-ta*, *vieraš-ta*, *valehut-ta*, *tois-ta*, *nais-ta*, *laisut-ta*, *yh-tä*, *terveht-tä*, *igähis-tä*, *inehmis-tä*, *jytyt-tä*, *miityt-tä*, *elävyt-tä*, *aigahis-tä*, *hevos-tä*, *kummas-tä*, *pagolas-tä*, *houkas-tä*). В последних пяти примерах наблюдается несоответствие в использовании передне- или заднерядного показателя партитива. В тексте памятника здесь (в заднерядных словах) употребляется переднерядный вариант аффикса *-та*, что влечет за собой нарушение рядного сингармонизма. В новописьменном варианте языка в целях соблюдения гармонии гласных в подобных случаях используется окончание *-tä*, содержащее знак смягчения.

Для образования форм **партитива множественного числа** в источнике используются окончания *-е*, *-а*, *-да*, распределение которых представлено следующим образом:

- 1) в позиции после словоизменительной основы множественного числа, заканчивающейся на краткий гласный *-i*, употребляется окончание партитива *-е*, напр., *елави-е* ‘живой’, *пюги-е* ‘священный’, *гади-е* ‘нужда’, *рйагки-е / рйахки-е* ‘грех’, *вѣраги-е* ‘чужой’, *пұхтаги-е* ‘чистый’, *оми-е* ‘свой’, *вангемби-е* ‘старший’, *гуонукки-е* ‘строение’, *колми-е* ‘три’, *генгеллизи-е* ‘душевный’, *кодилази-е* ‘домашний’, *кодигизи-е* ‘домочадец’, *рйагказизи-е* ‘грешный’, *тойзи-е* ‘другой’ (ср. литер. *elävi-e*, *pyhi-e*, *hädie*, *riähki-e*, *vierahi-e*, *puhtahi-e*, *omi-e*, *vanhemi-e*, *huonukši-e*, *kolmi-e*, *hengellizi-e*, *kodilazi-e*, *kodihizi-e*, *riähkähizi-e*, *toizi-e*);
- 2) в позиции после основы множественного числа на сужающийся дифтонг с первым компонентом *-o* используется окончание партитива *-а*, при этом второй компонент дифтонга может сливаться с йотированным показателем, напр., *абивойндо-а* ‘обида’, *герро-а* ‘господин’, *авко-а* ‘явка’, *юмало-а* ‘Бог’, *лйико-а* ‘икона’, *пайко-а* ‘место’, *шано-а* ‘слово’, *муаттоло-а / муаттоло-а* ‘отец’, *муамоло-а* ‘мать’ (ср. литер. *abivoindoi-da*, *herroi-da*, *javkoi-da*, *jumaloi-da*, *liikoi-da*, *paikkoi-da*, *šanoi-da*, *tuattoloi-da*, *muamoloi-da*);
- 3) в позиции после основы множественного числа на дифтонг *-ey* употребляется окончание партитива *-да*, напр., *куолей-да* ‘покойник’ (ср. литер. *kuolieloi-da* < *kuolie*).

Аналогичная схема образования форм партитива множественного числа характерна для собственно карельских диалектов Карелии, но не Центральной России. В тверских карельских говорах здесь преимущественно используются форманты *-е* и *-да / -дä* (= *-da / -dä*) (ID 2295 [14]), что нашло отражение в нормированном варианте языка. Зафиксированные в середине прошлого столетия примеры употребления окончания *-ja / -jä* (= *-a*) могут свидетельствовать в пользу постепенной унификации представленной в памятнике архаичной системы.

К **семантическим падежам**, имеющим единое окончание в единственном и множественном числах, относятся местные падежи с широким кругом значений и падежи с конкретным значением. **Местные падежи** по двигательному параметру делятся на падежи приближения (транслатив, иллатив, аллатив), удаления (элатив, аблатив) и нахождения (эссив, инессив, адессив). По ориентирующему – на внутрнеместные (инессив, элатив, иллатив), внешнеместные (адессив, аблатив, аллатив) и общеместные (эссив, транслатив).

Для собственно карельского наречия характерно совпадение падежных показателей адессива и аллатива, однако наличие контекста позволяет при необходимости развести их по значению. Падеж **адессив-аллатив** по числу употреблений в тексте перевода является самым частотным. Для его образования используются окончания:

- 1) **-лла / -лла**, выступающее в позиции после краткого гласного или расширяющегося дифтонга основы слова, в формах единственного: *пага-лла* 'плохой', *шана-лла* 'слово', *пұхтага-лла* 'чистый', *юмизе-лла* 'питье', *пуоле-лла* 'сторона', *малитү-лла* 'молитва', *мйіру-лла* 'мирра', *муамо-лла* 'мать', *туато-лла* 'отец', *пани-лла* 'свещенник', *мүа-лла* 'земля', *тожіе-лла* 'правда', *азіе-лла* 'дело', *гюва-лла* 'хороший', *кевга-лла* 'бедный', *тюгья-лла* 'пустой', *кіеле-лла* 'язык', *вере-лла* 'кровь', *міега-лла* 'муж', *генге-лла* 'душа', *ріагжагизе-лла* 'грешный', *гибіе-лла* 'тело' (ср. литер. *paha-lla*, *šana-lla*, *puhtaha-lla*, *juomize-lla*, *puole-lla*, *malitu-lla*, *miiru-lla*, *tuamo-lla*, *tuato-lla*, *papi-lla*, *mua-lla*, *tožie-lla*, *azie-lla*, *hyvä-llä*, *keyhä-llä*, *tyhjä-llä*, *kiele-llä*, *vere-llä*, *miehe-llä*, *henge-llä*, *hibie-llä*), а также множественного числа: *тойзи-лла* 'другой', *кевги-лла* 'бедный', *мйітюзи-лла* 'какой' (ср. литер. *toizi-lla*, *keyhi-llä*, *miittyzi-llä*);
- 2) **-ла / -ла**, следующее за основой на сужающийся дифтонг со вторым компонентом *-й*. Как правило, данный показатель используется в формах множественного числа: *пагой-ла* 'плохой', *геррой-ла* 'господин', *шаной-ла* 'слово', *велганіекой-ла* 'должник', *израстилой-ла* 'подобие', *руадолой-ла* 'работа', *туаттолой-ла* 'отец', *азей-ла* 'дело', *кашекенній-ла* 'заповедь' (ср. литер. *pahoi-lla*, *herroi-lla*, *šanoi-lla*, *velganiekoi-lla*, *izrastiloi-lla*, *ruadoloi-lla*, *tuattoloi-lla*, *azieloi-lla*, *käškennoi-llä*).

В диалектных данных, как и в нормированном варианте языка, отмечено использование единого варианта падежного показателя с геминированным согласным компонентом. Исключением являются формы личных и указательных местоимений с одиночным согласным в аффиксе во всех собственно карельских говорах (ID 1187, 1188 [14]), а также ряд северо-восточных толмачевских и валдайских говоров (ID 1182, 1184 [14]), в которых употребление этих алломорфов отмечено и в других классах имен [19]. Такое положение вещей в совокупности с данными памятника письменности указывает на архаичность показателя *-ла / -ла*, употреблявшегося исключительно в позиции после сужающегося дифтонга основы слова. Очевидно, его распространение шло из местоименных словоформ. Отмеченная схема распределения алломорфов сошла на нет в тверских говорах ко второй половине XX в.

Следующим по популярности падежом в тексте является **инессив**. Для его образования используются окончания **-иша / -иша**, **-сса**, **-са / -са**, следующие за слабоступенной основой имен. Распределение показателей представлено следующим образом:

- 1) окончание инессива **-иша / -иша** выступает в позиции после краткого гласного основы, если этот гласный не является гласным *-и*, напр., *тайвага-иша* 'небо', *кирьютукше-иша* 'писание', *вара-иша* 'страх', *вүүвве-иша* 'год', *пайка-иша* 'место', *кумбазе-иша* 'который', *колманне-иша* 'третий', *іа-иша* 'век', *тюгья-иша* 'пустое', *вее-иша* 'вода', *міеле-иша* 'мысль', *кирикю-иша* 'церковь', *еланна-иша* 'жизнь' (ср. литер. *taivaha-šša*, *kirjutukše-šša*, *vara-šša*, *vuuve-šša*, *paika-šša*, *kumbaze-šša*, *kolmanne-šša*, *ijä-ššä*, *tyhjä-ššä*, *veje-ššä*, *miele-ššä*, *kirikö-ššä*, *elännä-ššä*). Обращает на себя внимание отсутствие унификации при распределении передне- и заднерядных вариантов показателя, что, очевидно, вызвано артикуляционными характеристиками переднеязычного щелевого согласного окончания, напр., *тойзе-иша* 'второй', *тюгья-иша* 'пустой', *кашишенна-иша* 'заповедь', *міеле-иша* 'мысль', *вйіенне-иша* 'пятый', *югекшанне-иша* 'девятый', *шоотенна-иша* 'кормление' (ср. литер. *toize-šša*, *tyhjä-ššä*, *käššennä-ššä*, *miele-ššä*, *vijjenne-ššä*, *yhekšänne-ššä*, *šyötennä-ššä*);
- 2) показатель **-сса** используется после краткого *-и* основы слова, например, в формах множественного числа: *юкси-сса* 'один', *шилми-сса* 'глаз', *ализи-сса* 'нижний' (ср. литер. *yksi-ssä*, *šilmi-ssä*, *alizi-ssa*);

- 3) аффикс *-са / -са* употребляется в позиции после сужающегося дифтонга основы (см. дистрибуция показателей адессива-аллатива). Речь может идти как о формах единственного: *кой-са / кой-сса* ‘дом’ (ср. литер. *koi-ssa*), так и множественного числа: *пагой-са* ‘плохой’, *кирвйой-са* ‘книга’, *азей-са* ‘дело’, *омйй-са* ‘свой’, *присагой-са* ‘присяга’, *малитуллой-са* ‘молитва’, *кашшенийой-са* ‘заповедь’, *миеллйлой-са* ‘мысль’ (ср. литер. *pahoi-ssa, kirjoi-ssa, azieloi-ssa, omi-ssa, pris'agoi-ssa, malittuloi-ssa, käššennöi-ssä, mielilöi-ssä*).

Для образования **иллатива** использованы окончания **-хъ, -же**:

- 1) формант **-же** употребляется в двусловных именах с гласной основой на *-га-* (очевидно, также *-ге-*): *тайвага-же* ‘небо’ (ср. литер. *taivaha-h*);
- 2) окончание **-хъ**, присоединяющееся к сильной гласной основе, выступает в остальных словоизменительных типах имен: *пага-хъ* ‘плохой’, *айга-хъ* ‘время’, *пойга-хъ* ‘сын’, *илма-хъ* ‘свет’, *еланда-хъ* ‘жизнь’, *ниме-хъ* ‘имя’, *ведъ-хъ* ‘вода’, *юхте-хъ / юхтъ-хъ* ‘один’, *кириккйой-хъ / кириккйой-хъ* ‘церковь’, *туатто-хъ* ‘отец’, *пареннукше-хъ* ‘исправление’, *шүамизе-хъ* ‘приобретение’, *оттамизе-хъ* ‘применение’, *валгйе-хъ* ‘белый’, *сватой-хъ* ‘святой’ (ср. литер. *paha-h, aiga-h, poiga-h, ilma-h, eländä-h, nime-h, vede-h, yhte-h, kirikkö-h, tuatto-h, parennukše-h, šuamize-h, ottamize-h, valgie-h, sv'atoi-h*).

Закрепленные в тексте памятника архаичные формы иллатива на **-же** не характерны для словоизменительной системы тверских карельских говоров середины прошлого столетия (ID 1176, 1177 [14]), а, соответственно, не нашли отражения в новописьменном варианте языка.

Не демонстрирует отличий от современного представительства образование форм **эллатива** с окончаниями:

- 1) **-шта / -шта**, присоединяющимся к гласной основе (слабой для одноосновных имен), заканчивающейся на любой гласный кроме *-и* (*-й* в сужающихся дифтонгах), напр., *илма-шта* ‘свет’, *пага-шта* ‘плохой’, *вйеро-шта* ‘вера’, *туато-шта* ‘отец’, *тавде-шта* ‘болезнь’, *рагвага-шта* ‘народ’, *тайвага-шта* ‘небо’, *пиккурузе-шта* ‘маленький’, *оттамизе-шта* ‘применение’, *валгйе-шта* ‘белый’, *тавве-шта* ‘полный’ (ср. литер. *ilma-šta, paha-šta, viero-šta, tuato-šta, taive-šta, rahvaha-šta, taivaha-šta, pikkuruz-šta, ottamize-šta, valgie-šta, täyve-šta*). Как и в случае с инессивом, в позиции после шипящего согласного окончания возможно нарушение сингармонизма: *кириккйой-шта* ‘церковь’, *русситтамизе-шта* ‘крещение’, *тютйой-шта* ‘девочка’ (ср. литер. *kirikkö-šta, rissittämize-šta, tytö-šta*);
- 2) **-ста / -ста**, выступающим после гласного *-и* (*-й* в сужающихся дифтонгах) основы, как в формах единственного: *папи-ста* ‘священник’, *сватой-ста* ‘святой’ (ср. литер. *papi-sta, sv'atoi-sta*), так и множественного числа: *пагой-ста* ‘плохой’, *найзи-ста* ‘жена’, *кумбазы-ста* ‘который’, *кевгы-ста* ‘бедный’, *рйагы-ста / рйагй-ста* ‘грех’, *миеллйлой-ста* ‘мысль’ (ср. литер. *pahoi-sta, naizi-sta, kumbazi-sta, keyhi-stä, riähi-stä, mielilöi-stä*).

Редкие формы **аблатива**, для образования которых в переводе используется падежное окончание **-лда**, напр., *муаилма-лда* ‘мир’, *юмала-лда* ‘Бог’ (ср. литер. *muailma-lda, jumala-lda*), также демонстрируют совпадение с диалектными данными.

Для образования падежа состояния **эссива** в переводе употребляются форманты **-на / -на**, не требующие ослабления согласного компонента основы, напр., единственное число: *абунйекка-на* ‘помощник’, *руохтина-на* ‘Господь’, *колманде-на / колма-на* ‘третий’, *парага-на* ‘лучший’, *паремба-на* ‘более хороший’, *тожикага-на* ‘правдивый’, *пухтага-на* ‘чистый’, *вере-на* ‘кровь’, *гюва-на* ‘хороший’ (ср. литер. *abuniekka-na, ruohtina-na, kolmande-na, parahana-na, paremba-na, tožikkahana-na, puhtaha-na, vere-nä, huvä-nä*); множественное число: *пруазнйеккой-на* ‘праздник’ (ср. литер. *pruazniekkoi-na*). Следует отметить редкие примеры

нарушения закона гармонии гласных, напр., *гибіе-на* ‘тело’, *вешшела-на* ‘веселый’, *пайва-на* / *пйа-на* ‘день’ (ср. литер. *hibie-nä, veššela-nä, päivä-nä* / *piä-nä*).

Окончаниями **транслатива** выступают **-киши**, **-кси**, присоединяющиеся к гласной основе слова (слабой для одноосновных имен):

- 1) формант **-киши** следует за основами, заканчивающимися на любой гласный компонент кроме **-и** (**-й** в сужающихся дифтонгах): *пага-киши* ‘плохой’, *ристиканжа-киши* ‘человек’, *юмала-киши* ‘Бог’, *слава-киши* ‘живой’, *міеґе-киши* ‘мужчина’, *шүүре-киши* ‘великий’ (ср. литер. *paha-kši, ristikanža-kši, jumala-kši, elävä-kši, miehe-kši, šuure-kši*);
- 2) формант **-кси** присоединяется к основам на **-и** (**-й** в сужающихся дифтонгах). При этом речь может идти как о лексических основах единственного числа: *папи-кси* ‘священник’ (ср. литер. *papi-ksi*), так и о словоизменительных основах множественного числа: *енҗимази-кси* ‘первый’ (ср. литер. *enžimäzi-ksi*).

Образование транслатива также не обнаруживает отличий с современным представительством категорий.

Указывающий на отсутствие чего-либо падеж **абессив** формируется с помощью окончания **-тта** как в формах единственного: *гйа-тта* ‘нужда’, так и множественного числа: *гюви-тта* ‘хороший’ (ср. литер. *hiä-ttä, hyvi-ttä*). Обращает на себя внимание не характерное для современной диалектной речи и литературного языка явление сокращения геминаты показателя в позиции после сужающегося дифтонга словоизменительной основы множественного числа: *азе-й-та* ‘дело’ (ср. литер. *azieloi-tta*). В данном случае речь идет о явлении того же порядка, что и в формах инессива и адессива-аллатива.

Именная словоизменительная система языка памятника письменности не зафиксировала употребление падежей послеложного образования комитатива и аппроксиматива. Значение **комитатива**, выражающего сопроводителя действия и совместимость, в источнике передается послеложной конструкцией генитив + послелог *кера* ‘с’, например, ... *гибіень кера новҗи тайваґаҗе* ‘... с плотью вознесся на небо’, ... *луадіе паґуа найзҗинегмизень кера* ... ‘... совершать блуд с женщиной ...’, ... *кайкйень кера гювинь еліа* ... ‘... со всеми хорошо жить ...’, ... *пұхтаґынъ веень кера шевотеттұ*. ‘... с чистой водою соединенное...’ (ср. литер. *hibie-nke, naizinehmize-nke, kaikkie-nke, veje-nke*). В диалектных материалах середины XX в. в большей части карельских говоров Центральной России отмечено использование падежа комитатива с окончанием **-nke** (**-nken**, **-nkena**) (ID 1196, 1197 [14]). Распространение комитатива относится к древнекарельскому периоду, однако, полное отсутствие его фиксаций в памятниках тверской карельской письменности свидетельствует в пользу позднего самостоятельного характера послеложного образования.

Значение движения по направлению к кому-либо, чему-либо выражается при помощи послеложной конструкции генитив + *луохъ* ‘к’: ... *гйанъ омұа табұа мюотень мйанъ раґкаґіезень луохъ тұли тайваґаҗита* ... ‘... он по милосердию своему к нам грешным сошел с неба ...’, тогда как в северо-восточных тверских карельских говорах возможно использование падежа с окончанием **-lloo** / **-llyö** (ID 1201, 1203 [14]) (*riähkähize-llyö*). Тенденция к использованию послеложного падежа **аппроксиматива** на месте соответствующей послеложной конструкции в собственно карельских говорах Центральной России, не нашедшая отражения в тексте перевода, таким образом, также имеет внутренние истоки и может быть отнесена к поздним инновациям.

4. Словоизменение местоимений

В тексте перевода представлены местоимения следующих разрядов:

- 1) личные: *міе* ‘я’, *шіе* ‘ты’, *гйанъ* ‘он, она, оно’, *мюіо* / *мюо* ‘мы’, *гюо* ‘они’;
- 2) указательные: *тама* ‘этот, эта, это’, *тұо* ‘тот, та, то’, *же* ‘тот, та, то’, *нама* ‘эти’;
- 3) притяжательное: *ома* ‘свой, собственный’;

- 4) возвратное: *иче* ‘сам’;
- 5) обобщительно-определятельные: *юго* / *юго* / *юго* ‘каждый’, *югогине* ‘каждый, всякий’, *кайки* ‘все’, *молеммать* ‘оба’, *тойне* ‘другой’, *юкси* ‘только один’;
- 6) вопросительные / относительные: *ми* ‘что’, *кенъ* ‘кто’, *кумбане* ‘который’, *мйитюшь* ‘какой’;
- 7) отрицательные: *ними* ‘ничто’, *ни кенъ* ‘никто’, *ни мйитюшь* ‘никакой’;
- 8) неопределенные: *ми нибуйтень* ‘что-нибудь’, *мйитюшь нибуйтень* ‘какой-нибудь’.

В карельском языке, как и в других прибалтийско-финских языках, местоимения наделены особыми архаичными словоизменительными парадигмами, отличными от остальных имен. Речь идет не только об особенностях лексических и словоизменительных основ, но и о падежных показателях, употребляемых в отдельных словоформах [20]. Это касается, главным образом, первостепенных по происхождению личных, указательных и вопросительных местоимений (табл. 2, 3). Остальные местоимения, встречающиеся в тексте перевода, склоняются также как и имена соответствующего словоизменительного типа.

Табл. 2. Склонение личных местоимений.

Table 2. Declension of personal pronouns.

	‘я’	‘ты’	‘он, она, оно’	‘мы’	‘они’
Номинатив	<i>міе</i>	<i>Шіе</i>	<i>гїанъ</i>	<i>мюо / мюіо</i>	<i>гюо</i>
Генитив	<i>мівнъ / мівнъ</i>	<i>Шивнъ</i>	<i>ганенъ</i>	<i>міанъ / міанъ / мианъ</i>	<i>гїанъ</i>
Партитив	<i>милма</i>	<i>Шилма</i>	<i>ганда</i>	<i>мейда</i>	<i>гейда / геїда</i>
Элатив			<i>ганешта</i>		
Адессив	<i>мівла / мивла</i>	<i>Шивла</i>	<i>ганелла</i>	<i>мейла</i>	<i>гейла</i>

Табл. 3. Склонение указательных и вопросительных местоимений.

Table 3. Declension of demonstrative and interrogative pronouns.

	‘этот, эта, это’	‘тот, та, то’	‘тот, та, то’	‘эти’	‘кто’	‘что’
Номинатив	<i>тама</i>		<i>же</i>	<i>нама</i>	<i>кенъ</i>	<i>ми</i>
Генитив	<i>таманъ</i>		<i>женъ</i>	<i>найїенъ</i>	<i>кененъ</i>	<i>минъ</i>
Партитив	<i>тада</i>		<i>шида</i>	<i>найда</i>	<i>кѣда / кѣда</i>	<i>міда / мида</i>
Инессив				<i>найса</i>		
Элатив					<i>кешта</i>	
Иллатив			<i>шихъ</i>			<i>миг / михъ</i>
Адессив	<i>талла</i>	<i>туолла</i>	<i>шилла</i>		<i>келла</i>	
Аблатив					<i>келда</i>	

В современных говорах (ID 1116, 1119, 1122, 1136, 1138, 1142, 1174, 1187, 1188 [14]), как и в нормированном варианте тверского карельского языка, система склонения местоимений практически не отличается от представленной в памятнике (но *найїенъ* ‘эти-GEN’ > *näin*), что подтверждает ее устойчивый характер.

5. Категория притяжательности

В карельском языке для выражения притяжательности, наряду с генитивными конструкциями и морфологически неоформленными типами посессивных словосочетаний, используются притяжательные суффиксы. В современных тверских карельских говорах они практически вышли из употребления.

В тексте анализируемого источника зафиксировано 19 примеров использования притяжательных суффиксов второго и третьего лица.

Посессивным суффиксом второго лица единственного числа является *-ишъ*: *иче-лла-ишъ* ‘сам-ALL-POSS.2SG’, *туатту-о-ишъ* ‘отец-PRT-POSS.2SG’, *муаму-о-ишъ* ‘мать-PRT-POSS.2SG’.

Например, *Сла луаи ичеллашъ израстѣ ниминъ ...* ‘Не сотвори тебе кумира ...’, *Кууңделе туаттуошъ и муамуошъ шивнъ ...* ‘Чти отца твоего и мать твою ...’.

Для выражения третьего лица используется притяжательный суффикс *-хъ*. Примеры образования данной формы ограничены исключительно местоимениями: *кайкѣ-е-хъ* ‘все-*PRT-POSS.3SG*’, *ган-да-хъ* ‘он-*PRT-POSS.3SG*’, *гей-ла-хъ* ‘они-*ALL-POSS.3SG*’, *гей-да-хъ* ‘они-*PRT-POSS.3SG*’. Например, ... *ганда юхта кайкѣхъ паразиттуанъ шувата и кууңелла ...* ‘... его единого больше всех любить и почитать ...’, ... *пидавъ гандахъ шувата кайкѣлла генгелла ...* ‘... должно Его любить сердечно ...’, ... *аннанъ тѣдѣл гейлахъ ...* ‘... дам знать им ...’.

Представленная картина позволяет заключить, что архаичная категория притяжательности уже в языке тверских карелов периода начала XIX в. не отличалась своей продуктивностью и постепенно вытеснялась генитивными конструкциями, напр., *ливнъ муамъ* ‘моя мать’, *юмала шивнъ* ‘Бог твой’, ... *ни кюлѣ ганенъ, ни инегмиста ганенъ, ни гаргѣа ганенъ, ни гевоста ганенъ ...* ‘... ни села его, ни раба его, ни быка его, ни коня его ...’, *нимѣ ганенъ* ‘его имя’, *гѣанъ нименъ алла* ‘под именем их’, *каннеттулоенъ гѣанъ* ‘их детей’.

6. Заключение

Анализ именной словоизменительной системы языка анализируемого памятника карельской письменности 1804 г. в сравнении с диалектными данными середины XX в. и новописьменным вариантом языка позволил выделить ряд архаичных и инновационных особенностей, отличающих ее как от современной, так и от реконструируемой древнекарельской.

К архаичным явлениям, не нашедшим продолжения в современных говорах Центральной России, относятся: использование показателя множественного числа *-й* с многосложными одноосновными именами на дифтонг; окончание множественного числа генитива *-енъ*; окончание множественного числа партитива *-а*; окончания адессива-аллатива *-ла* / *-ла*, инессива *-са* / *-са* и абессива *-та*; окончание иллатива *-же*; использование послеложных конструкций для передачи значений комитатива и аппроксиматива; притяжательные суффиксы второго и третьего лица единственного числа.

Употребление притяжательных суффиксов в языке анализируемого памятника – довольно редкое явление, что может указывать на инновационную тенденцию перехода от посессивных суффиксов к генитивным конструкциям. Инновационный характер демонстрирует также распространение форманта множественности *-лой-* / *-лѣй-* на одноосновные имена с гласной основой на *-е*.

Именная словоизменительная система памятника, за исключением выделенных архаичных явлений, не демонстрирует существенных отличий от современных тверских говоров собственно карельского наречия. Это подтверждает гипотезу о диалектной основе языка-источника памятника. В пользу данного вывода свидетельствует также совпадение его именной морфологии с именной словоизменительной системой языка перевода «Евангелия от Марка», выполненного на толмачевские говоры в 1820 г. священниками Г. Е. Введенским и М. А. Золотинским [21].

Список литературы / References

- [1]. Прокопьев К. Переводы христианских книг на инородческие языки в первой половине XIX в. Казань, Типо-лит. Имп. Ун-та, 1904, 40 с. / Prokopiev K. Translations of Christian books into foreign languages in the first half of the 19th century. Kazan, Typo-lithography of the Imperial University, 1904, 40 p. (in Russian).
- [2]. Переводъ нѣкоторыхъ молитвъ и сокращеннаго катихизиса на Корельскій языкъ. Санктпетербург, 1804, 31 с. / Translation of some prayers and the shorter catechism in the Korelian language. St. Petersburg, 1804, 31 p. (in Russian).

- [3]. Памятники лексикографии в «Цветнике» Прохора Коломнятина 1668 года. Исследования и тексты. М.; СПб., Альянс-Архео, 2023, 634 с. / Monuments of lexicography in Prokhor Kolomnyatin's "Flowerbed" of 1668. Research and texts. Moscow, Saint Petersburg, Alliance-Archeo, 2023, 634 p. (in Russian).
- [4]. Новак. И. П. Определение диалектной специфики карельских материалов словаря П. С. Палласа. Финно-угорский мир, №1, 2024, стр. 33–49. DOI: 10.15507/2076-2577.016.2024.01.33-49. / Novak I. P. Definition of dialect specificity of Karelian materials of P. S. Pallas' dictionary. Finno-Ugric world. 2024, No 1, pp. 33–49 (in Russian). DOI: 10.15507/2076-2577.016.2024.01.33-49.
- [5]. Громова Л. Г. Особенности развития карельской письменности на толмачевском говоре тверского диалекта. Тверские карелы: история, язык, культура. Тверь, Твер. гос. ун-т, 2011, стр. 42–54. / Gromova L. G. Features of the development of Karelian writing in the Tolmachevo dialect of the Tver dialect. Tver Karelians: history, language, culture. Tver, Tver State University, 2011, pp. 42–54 (in Russian).
- [6]. Ison suomen kieliopin verkkoversio. Available at: <https://kaino.kotus.fi/visk/etusivu.php> (accessed 30.05.2025).
- [7]. LingvoDoc: лингвистическая платформа. Доступно по ссылке: <https://lingvodoc.ispras.ru/> (дата обращения 20.05.2025). / LingvoDoc: linguistic platform. Available at: <https://lingvodoc.ispras.ru/> (accessed 20.05.2025).
- [8]. Перевод некоторых молитв и сокращенного катехизиса на карельский язык: glossированный текст. Доступно по ссылке: <https://lingvodoc.ispras.ru/dictionary/11306/17/perspective/11306/21/view> (дата обращения 22.05.2025). / Translation of some prayers and the abbreviated catechism into Karelian: glossed text. Available at: <https://lingvodoc.ispras.ru/dictionary/11306/17/perspective/11306/21/view> (accessed 22.05.2025).
- [9]. Словарь по памятнику письменности "Перевод некоторых молитв и сокращенного катехизиса на карельский язык". Доступно по ссылке: <https://lingvodoc.ispras.ru/dictionary/11429/2/perspective/11429/3/view> (дата обращения 23.05.2025). / Dictionary on the written monument "Translation of some prayers and the abbreviated catechism into the Karelian language". Available at: <https://lingvodoc.ispras.ru/dictionary/11429/2/perspective/11429/3/view> (accessed 23.05.2025).
- [10]. Морфологический словарь памятника письменности "Перевод некоторых молитв и сокращенного катехизиса на карельский язык". Доступно по ссылке: <https://lingvodoc.ispras.ru/dictionary/11430/2/perspective/11430/3/view> (дата обращения 24.05.2025). / Morphological dictionary of the written monument "Translation of some prayers and the abbreviated catechism into the Karelian language". Available at: <https://lingvodoc.ispras.ru/dictionary/11430/2/perspective/11430/3/view> (accessed 24.05.2025).
- [11]. Gromova. L. Terveh! Terveh!: tverinkarielan paginakursa. Петрозаводск, КарНЦ РАН, 2022, 226 с. / Gromova. L. Terveh! Terveh!: tverinkarielan paginakursa. Petrozavodsk, KarRC RAS, 2022, 226 p. (in Karelian).
- [12]. Murreh: диалектная база карельского языка. Доступно по ссылке: <http://murreh.krc.karelia.ru/> (дата обращения 25.05.2025). / Murreh: dialect base of the Karelian language. Available at: <http://murreh.krc.karelia.ru/> (accessed 25.05.2025) (in Karelian).
- [13]. Программы по собиранию материала для диалектологического атласа карельского языка // Научный архив КарНЦ РАН. Ф. 1. Оп. 5. №№ 79–94, 105–141, 160–166. Петрозаводск, 1957–1959. / Programs for collecting material for a dialectological atlas of the Karelian language. In: SA KRC RAS. F. 1. I. 5. Nos. 79–94, 105–141, 160–166. Petrozavodsk, 1957–1959 (in Karelian).
- [14]. Раздел «Вопросы» диалектной базы карельского языка Murreh. Доступно по ссылке: <http://murreh.krc.karelia.ru/ques/question> (дата обращения 25.05.2025). / Section "Questions" of the dialect database of the Karelian language Murreh. Available at: <http://murreh.krc.karelia.ru/ques/question> (accessed 25.05.2025) (in Karelian).
- [15]. Leskinen E. Tulemajärven murteen vokalismi. Helsinki, SKS, 1933, 138 s.
- [16]. Leskinen H. Karjala ja karjalaiset kielentutkimuksen näkökulmasta. Karjala: historia, kansa, kulttuuri. Helsinki, SKS, 1998, s. 352–382.
- [17]. Новак И. П. Карельский языковой ландшафт в диалектометрической парадигме. Петрозаводск, КарНЦ РАН, 2025, 502 с. / Novak I. P. Karelian linguistic landscape in the dialectometric paradigm. Petrozavodsk, KarRC RAS, 2025, 502p. (in Russian).
- [18]. Пунжина А. В. Именные категории в калининских говорах карельского языка. Дис. ... канд. филол. наук. Петрозаводск, 1975, 108 с. / Punzhina A. V. Nominal categories in the Kalinin dialects of the Karelian language. Diss. ... Cand. Philological Sciences. Petrozavodsk, 1975, 108 p. (in Russian).

- [19]. Palmeos P. *Karjala valdai murrak*. Tallinn, Tartu Ülikooli, 1962, 226 s.
- [20]. Гилоева Н. М. Вопросительные, неопределенные и обобщительно-определятельные местоимения в диалектах карельского языка. Дис. ... канд. филол. наук. Петрозаводск, 2004, стр. 6. / Giloeva N. M. Interrogative, indefinite and generalizing-definitive pronouns in the dialects of the Karelian language. Diss. ... Cand. Philological Sciences. Petrozavodsk, 2004, p. 6 (in Russian).
- [21]. Новак И. П. Именная морфология памятника тверской карельской письменности – перевода «Евангелия от Марка». Вестник Новосибирского государственного университета. Серия: История, филология. 2022, № 9, стр. 9–20. DOI: 10.25205/1818-7919-2022-21-9-9-20. / Novak I. P. Nominal morphology of a monument of Tver Karelian writing – the translation of the “Gospel of Mark”. Bulletin of the Novosibirsk State University. Series: History, Philology. 2022, No 9, pp. 9–20 (in Russian). DOI: 10.25205/1818-7919-2022-21-9-9-20.

Информация об авторах / Information about authors

Ирина Петровна НОВАК – доктор филологических наук, старший научный сотрудник Междисциплинарного научно-образовательного центра Nordica Института языка, литературы и истории Карельского научного центра РАН, ведущий научный сотрудник лаборатории «Лингвистические платформы» Института системного программирования РАН. Сфера научных интересов: корпусная лингвистика, компьютерная лингвистика, диалектология.

Irina Petrovna NOVAK – Dr. Sci. (Philology), Senior Researcher at the Nordica International Scientific and Educational Center of the Institute of Language, Literature and History of the Karelian Research Center of the Russian Academy of Sciences, Leading Researcher at the Linguistic Platforms Laboratory of the Institute of System Programming of the Russian Academy of Sciences. Research interests: corpus linguistics, computational linguistics, dialectology.



Управление батареей электромобиля на основе машинного обучения и ПЛИС

¹ Р.Д. Мери́на, ORCID: 0009-0005-3452-2303 <rdaisymerina@gmail.com>

² Р.С. Ра́м, ORCID: 0000-0002-1132-2975 <saravanaramkrishnan@gmail.com>

³ Л.С.П. Ма́йкл, ORCID: 0000-0001-9867-6276 <drlordwin@veltech.edu.in>

¹ Вел Тех МультиТех Инженерный колледж докторов Сакунтхала и Рангараджана, Индия, Ченнай.

² Факультет электроники и связи, Университет Анны – Мадурайский региональный кампус, Индия, Мадурай.

³ Вел Тех МультиТех Институт науки и технологий доктора Сакунтхала и Рангараджана, Индия, Ченнай.

Аннотация. В работе представлена ПЛИС-реализация адаптивного блока управления питанием для электромобилей, основанная на гибридной модели с использованием двунаправленной долгосрочной краткосрочной памяти и дерева решений. Разработка выполнена на платформе Zynq UltraScale+ MPSoC и предназначена для оценки состояния заряда аккумулятора и классификации условий вождения с целью оптимального распределения мощности между компонентами электроавтомобиля. Аппаратный стенд моделирует характеристики среднего электромобиля, генерируя данные о положении дросселя, скорости, напряжении/токе батареи и координатах каждые 30 секунд. Экспериментальные результаты показывают значительное улучшение энергоэффективности и сокращение задержки по сравнению с традиционными блоками управления батареями. Предложенная система потребляет около 0,98 Вт, обеспечивает задержку 5,6 мкс и достигает производительности 181,6 операций на ватт, значительно превосходя устройства с микроконтроллерами или сигнальными процессорами. Расчет запаса хода демонстрирует увеличение до 25% в условиях шоссе при использовании модели Bi-LSTM + дерево решений, что подтверждает эффективность предлагаемой адаптивной стратегии управления энергетикой электроавтомобиля.

Ключевые слова: система управления батареями для электромобилей; встраиваемая реализация на базе ПЛИС; рекуррентные нейронные сети с долгой краткосрочной памятью; двунаправленная архитектура Bi-LSTM; классификация с помощью дерева решений; алгоритмы оценки уровня заряда; распознавание стиля вождения с использованием машинного обучения.

Для цитирования: Мери́на Р.Д., Ра́м Р.С., Ма́йкл Л.С.П. Управление батареей электромобиля на основе машинного обучения и ПЛИС. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 215–232. DOI: 10.15514/ISPRAS-2025-37(4)-13.

Machine Learning-Enabled Battery Management System on FPGA for Electric Vehicles

¹R.D. Merina, ORCID: 0009-0005-3452-2303 <rdaisymerina@gmail.com>

²R.S. Ram, ORCID: 0000-0002-1132-2975 <saravanaramkrishnan@gmail.com>

³L.C.P. Micheal, ORCID: 0000-0001-9867-6276 <dlordwin@veltech.edu.in>

¹ Vel Tech Multi Tech Dr. Rangarajan Dr. Sakunthala Engineering College, Chennai, India.

² Department of Electronics and Communication Engineering,
Anna University – Regional Campus Madurai, India.

³ Department of Electronics and Communication Engineering,
Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, Chennai, India.

Abstract. This work implements a field-programmable gate array (FPGA) architecture for an intelligent battery management unit (i-BMU) designed for an electric vehicle (EV) and investigates how it can increase the mileage of the same. The unique feature of the suggested i-BMU is the development of an FPGA architecture to allocate power to various electrical and electronic components of an EV by considering the run-time driving pattern and the state of charge (SoC) of the Li-ion battery. The proposed methodology involves dynamically estimating the SoC of the Li-ion battery using a Long Short-Term Memory Neural Network (LSTM-NN) model while the vehicle is in motion, predicting different driving cycles such as urban, highway, and downhill in real-time using a regression tree algorithm, and intelligently allocating electric power to various EV components based on the predicted driving cycle and the estimated SoC using a proposed power distribution algorithm. The proposed system is designed on the Zynq Ultrascale+ MPSoC development board, and the data given to the system for verification is collected through simulation of various sensor values for an electric bike, such as speed, throttle position, battery voltage, battery current, and GPS coordinates, by generating random data within typical operational ranges. The proposed system is compared with the existing system in terms of chip power consumption (W), area of the chip (mm²), computation time (μs), and throughput. Additionally, the suggested method evaluates the mileage of the EVs and extends their range by 17 km to 36 km depending on the driving pattern.

Keywords: battery management unit; long short-term memory neural network model (LSTM-NN); regression tree algorithm; field-programmable gate array; power distribution; driving cycle; state of charge (SoC).

For citation: Merina R.D., Ram R.S., Micheal L.C.P. Machine Learning-Enabled Battery Management System on FPGA for Electric Vehicles. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 215-232 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-13.

1. Введение

Электрические автомобили (ЭА) представляют собой революционный сдвиг в глобальной автомобильной промышленности, обеспечивая значительные преимущества с точки зрения экологической устойчивости, энергоэффективности и снижения долгосрочных эксплуатационных затрат. В условиях растущей необходимости сокращения выбросов парниковых газов и перехода к более чистым источникам энергии, ЭА рассматриваются как ключевой компонент транспортной системы будущего. Помимо нулевых выбросов вредных веществ и снижения зависимости от ископаемого топлива, они способствуют улучшению качества воздуха в городах и снижению шумового загрязнения. Несмотря на относительно высокую первоначальную стоимость, особенно в части аккумуляторных технологий, постоянный прогресс в области литий-ионных и твердотельных батарей, а также экономия за счёт масштаба производства, ведут к уменьшению разрыва в стоимости между электрическими и традиционными транспортными средствами.

Современные технологии ЭА включают в себя аккумуляторные электрические автомобили (АА), подключаемые гибридные автомобили (ПГЭА), гибридные автомобили (ГЭА) и автомобили на топливных элементах (АТЭ), каждая из которых имеет свою архитектуру и

способы получения энергии, соответствующие определённым сценариям эксплуатации. Производительность и запас хода любого электрического транспортного средства напрямую зависят от структуры и управления аккумуляторной системой. Основные подсистемы включают электрический двигатель, батарейный блок, блок управления силовой электроникой, встроенное зарядное устройство и систему теплового управления. Ключевым компонентом является система управления батареями (СУБ), которая отвечает за контроль уровня заряда (state of charge, SoC), температуры, напряжения и тока, обеспечивая безопасность и долговечность эксплуатации. Однако традиционные подходы к оценке SoC, основанные на эквивалентных схемах или электрохимических моделях, часто не справляются с высокой нелинейностью процессов и изменчивостью условий реального вождения, что снижает точность и надёжность оценок.

Цель данного исследования – разработка адаптивной интеллектуальной системы управления аккумуляторной батареей в режиме реального времени, использующей методы глубокого обучения и регрессионного анализа для повышения точности оценки SoC и обеспечения интеллектуального распределения энергии в ЭА. С этой целью предлагается инновационная архитектура, сочетающая двунаправленную нейронную сеть с долговременной краткосрочной памятью (Bidirectional long short-term memory, Bi-LSTM) и программируемую логическую интегральную схему (ПЛИС), а также регрессионное дерево принятия решений (regression decision tree, DT) для обеспечения объяснимости и интерпретируемости результатов оценки. Модель Bi-LSTM обучается на временных рядах данных от нескольких датчиков автомобиля с целью точного прогнозирования уровня заряда с учётом направлений потока данных как из прошлого, так и из будущего относительно текущего состояния. Для повышения точности предсказания в условиях сложных профилей движения дополнительно применяется регрессионное дерево, которое анализирует выходные параметры модели Bi-LSTM и классифицирует характер эксплуатационного режима в соответствии с текущими условиями. Кроме того, система учитывает стиль вождения, распознавая городские, шоссе и наклонные участки движения на основе данных от датчиков положения дроссельной заслонки и датчиков скорости на эффекте Холла.

Предложенная архитектура с ПЛИС-реализацией обеспечивает высокую скорость вычислений, масштабируемость и энергоэффективность, что критически важно для автомобильных приложений в реальном времени. Интеграция методов машинного обучения позволяет системе не только повышать точность оценки SoC по сравнению с классическими методами, но и учитывать дегенерационные процессы аккумулятора, адаптируясь к различным режимам эксплуатации. Это способствует более рациональному распределению энергии между различными компонентами транспортного средства и повышает общую надёжность силовой установки.

Система управления батареями от STMicroelectronics использует микроконтроллер ST-L9963 для защиты и оптимизации работы батареи, включая балансировку ячеек литий-ионных батарей [1]. В работе [2] представлена система оценки состояния заряда на основе генетического алгоритма, которая повышает точность оценки на 7,2% в сложных условиях. Исследование [3] анализирует проблемы и решения для зарядных станций с учетом географических данных и препятствий, улучшая доступность зарядных станций для владельцев электромотоциклов. Разработана усовершенствованная система управления батареями на базе ПЛИС с использованием MATLAB-FPGA, позволяющая точно прогнозировать УЗ и СА батареи и обеспечивающая низкое энергопотребление и высокую производительность [4]. В работе [5] обсуждаются проблемы балансировки элементов и моделирования в СУБ, а также предложены решения для устранения существующих недостатков.

Несмотря на достижения, точная оценка времени работы аккумуляторов остаётся сложной задачей. В исследованиях [6-7] предложены методы для улучшения оценки УЗ с учётом

нелинейных характеристик и факторов внешней среды. В работе [8] рассматривается метод, основанный на коррекции зависимостей, который может помочь в разработке безопасных и энергоэффективных систем управления батареями. Таким образом, многие существующие подходы к оценке УЗ не обеспечивают достаточной точности в реальных условиях, требующих учёта влияния старения батареи, циклов вождения и нестабильных внешних факторов [9]. Современные методы требуют дальнейших улучшений для повышения точности и адаптивности в динамичных условиях эксплуатации электромобилей.

Статья организована следующим образом. В Разделе 2 описывается архитектура предлагаемой интеллектуальной системы, включая аппаратную реализацию и обоснование выбора модели LSTM и ПЛИС. Раздел 3 посвящён методологии обучения нейронной сети, структуре данных и интеграции модели. В Разделе 4 приводятся результаты моделирования, сравнительный анализ и обсуждение ключевых метрик эффективности, таких как ошибка оценки, энергетическая эффективность и время отклика. Наконец, в Разделе 5 подводятся итоги исследования, обсуждаются ограничения текущей модели и намечаются перспективные направления для дальнейших разработок интеллектуальных систем управления энергией в электромобилях нового поколения.

2. Архитектура предлагаемой интеллектуальной системы управления батареями

Предложенная интеллектуальная архитектура распределения мощности использует передовые технологии искусственного интеллекта для динамического управления энергией в электромобилях (ЭА), с особым акцентом на оптимальное использование батареи и распределение мощности в реальном времени. Центральным элементом этой архитектуры является двунаправленная нейронная сеть с долговременной краткосрочной памятью (Bi-LSTM-NN), используемая для точной оценки состояния заряда (SoC) системы литий-ионных батарей, как показано на рис. 1 [10-12]. Bi-LSTM-NN захватывает как прошлые, так и будущие временные зависимости из входных последовательностей, таких как положение дроссельной заслонки, скорость автомобиля и предыдущие значения SoC, обеспечивая высокоточную и адаптивную оценку в различных условиях вождения. Эта двунаправленная структура улучшает способность к обучению по сравнению с традиционными однонаправленными сетями, что делает её особенно эффективной для приложений с последовательными временными рядами и быстро меняющимися требованиями нагрузки.

В дополнение к оценщику SoC, интегрирован модуль классификации цикла движения на основе дерева решений, который классифицирует рабочее окружение автомобиля на три типа: городской, шоссе и спуск. Эта классификация основана на обучающих наборах данных, которые содержат уникальные паттерны ускорения-замедления, средние скорости и профили потребления энергии, соответствующие каждому циклу [13]. Оригинальный набор данных D_1 и его обновленные варианты (D_2 и D_N) позволяют модели адаптироваться к новым стилям вождения и уточнять классификацию в реальном времени. Точно определяя цикл движения, система динамически корректирует стратегии распределения мощности, обеспечивая повышенную эффективность, снижение потерь энергии и улучшение дальности хода для ЭА. Выходная мощность P_{out} , необходимая от батареи, прямо пропорциональна решенной мощности $P_{decided}$, которая состоит из набора компонентов, специфичных для нагрузки: P_{PMSM_d} , P_{AC_d} , $P_{telemetry_d}$, P_{ECU_d} и P_{brake_d} . Эти компоненты вычисляются интеллектуальным алгоритмом с использованием комбинации фактического требования нагрузки и классифицированного цикла движения.

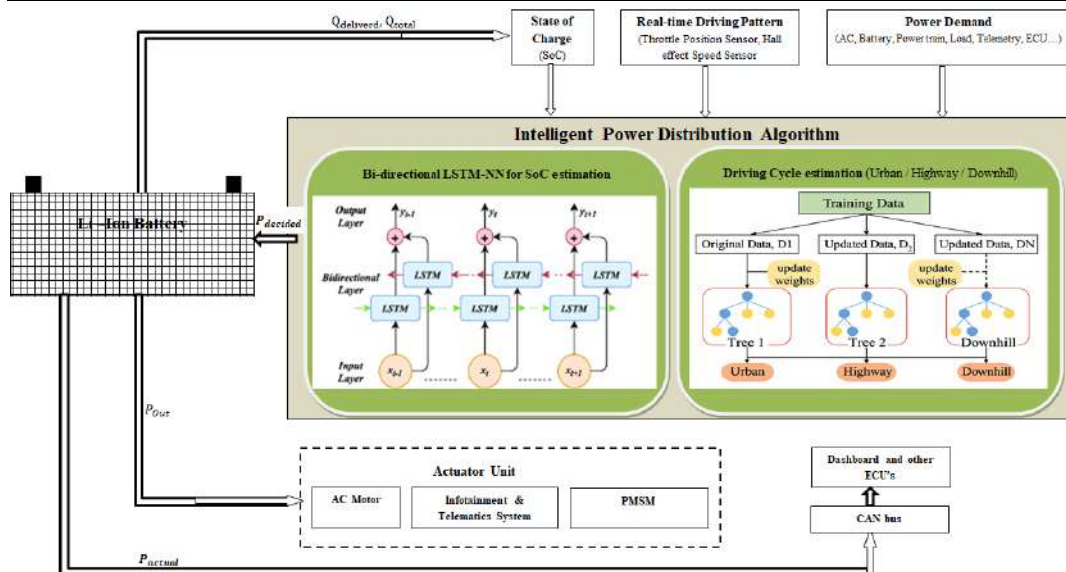


Рис. 1. Предлагаемая архитектура интеллектуального блока управления аккумулятором.

Fig. 1. Proposed architecture of an intelligent battery management unit.

3. Предложенный адаптивный алгоритм управления мощностью в динамических условиях эксплуатации

Предложенный Алгоритм 1 динамически оценивает состояние заряда (SoC) аккумулятора электромобиля (ЭА) и адаптивно управляет подачей энергии в ключевые электрические компоненты на основе условий вождения в реальном времени. Эта система синергетически интегрирует двунаправленную нейронную сеть с долговременной краткосрочной памятью (Bi-LSTM) для точной оценки SoC и классификатор дерева решений для категоризации режима вождения. Основная цель продлить срок службы батареи, повысить эксплуатационную безопасность и оптимизировать потребление энергии для различных подсистем автомобиля в условиях различных типов местности и нагрузки. При включении система инициализируется, распределяя предопределённое количество электрической мощности между каждым критически важным компонентом ЭА, основываясь на эксплуатационных требованиях и историческом использовании.

Эта начальная конфигурация мощности необходима для того, чтобы система стартовала в известном и безопасном состоянии, прежде чем адаптивное управление возьмёт на себя управление. Табл. 1 подытоживает начальную мощность, назначенную каждому компоненту. Этот начальный вектор записывается формулой (1).

$$P_{decided}^{initial} = \{P_{PMSM}^{initial}, P_{AC}^{initial}, P_{ECU}^{initial}, P_{infotainment}^{initial}, P_{lights}^{initial}\} \quad (1)$$

В предложенной структуре оценка состояния заряда (SoC) достигается с помощью двунаправленной нейронной сети с долговременной краткосрочной памятью (Bi-LSTM). В отличие от традиционных моделей SoC, которые опираются исключительно на методы подсчёта Кулона или напряжение при разомкнутой цепи, Bi-LSTM использует историческую и будущую контекстную информацию для улучшения точности предсказаний. Модель получает на вход последовательность векторов, включающих фактический заряд $Q_{delivered}$, общую ёмкость Q_{total} , ток $I(t)$ и напряжение $V(t)$ батареи, нормализованные как $x_t \in R^n$. Эти последовательности обрабатываются в обоих временных направлениях с помощью обратных

$\vec{h}_t$ и прямых $\vec{h}_t$ LSTM-единиц. Совмещённый вывод передается через слой регрессии, чтобы получить уравнение (2).

$$\widehat{SOC}_t = \text{ReLU}(W \cdot [\vec{h}_t; \vec{h}_t] + b) \tag{2}$$

В этом уравнении $\widehat{SOC}_t$ это оценка SoC в момент времени t , W – матрица весов, b – смещение, а ReLU обеспечивает неотрицательность. Сеть обучается с использованием функции потерь среднего квадратичного отклонения: $\mathcal{L}_{MSE} = \frac{1}{N} \sum_{t=1}^N (\widehat{SOC}_t - SOC_t^{actua})^2$, минимизируя отклонения от истинных значений SoC. Этот метод позволяет в реальном времени с высокой точностью предсказывать SoC, что важно для принятия адаптивных решений по распределению энергии.

```
BEGIN
// Inputs: Q_actual, Q_total, TP, Hall_sensor, P_optimum
// Output: P_decided to all EV components
SoC = Bi_LSTM_Estimate_SoC (Q_actual, Q_total)
Initialize P_decided with maximum device power values
WHILE SoC >= 40% DO
    drive_condition = Decision_Tree_Classify(TP, Hall_sensor)
    P_decided = Get_Power_By_Condition(drive_condition)
    SoC = Update_SoC(P_decided)
END WHILE
WHILE 10% < SoC < 40% DO
    P_decided = Assign_Minimum_Power()
    Trigger_Low_Battery_Alarm()
    Detect_Charging_Station()
    SoC = Update_SoC(P_decided)
END WHILE
IF SoC <= 10% THEN
    Enter_Low_Power_Mode()
    Trigger_Low_Battery_Alarm()
END IF
END
```

Алгоритм 1. Псевдокод предлагаемого метода распределения мощности.
Algorithm 1. Pseudo code for the proposed power distribution technique.

Табл. 1. Начальная электрическая мощность (Вт/ч), назначенная различным электрическим компонентам ЭА.

Table 1. Default electric power (Watts/hr) assigned to various electrical components of the EV.

Электрические компоненты	Электрическая мощность по умолчанию (Вт/ч)
Воздушный охладитель	3000
Фары (лампы HID+)	35
Информационно-развлекательная система	600
Электронный блок управления	600
Двигатель	3000

Для обеспечения контекстно-осведомленного распределения энергии, предложенная система классифицирует схемы вождения, а именно шоссеый, городской и спусковой режимы, используя классификатор дерева решений. Модель обрабатывает входные данные, такие как положение дроссельной заслонки, скорость двигателя с датчиков Холла и потребность в мощности в реальном времени $P_{optimum}$. Эти данные образуют вектор признаков $X=[\theta, \omega, P_{optimum}]$, где θ – угол дроссельной заслонки, а ω – угловая скорость. Дерево решений применяет рекурсивные бинарные разбиения на основе минимизации энтропии, что приводит к результату классификации $d \in \{0, 1, 2\}$, представляющему соответственно

шоссейный, городской и спусковой режимы. Например: если $\theta > \theta_{thres} \Rightarrow d=0$ (шоссе); если $\omega < \omega_{thres} \Rightarrow d=2$ (спуск); иначе $d = 1$ (город). Эта классификация напрямую влияет на логику распределения мощности, позволяя системе настраивать использование энергии в зависимости от местности и динамики движения. После определения SoC и условия вождения, модуль адаптивного управления мощностью перераспределяет мощность между компонентами ЭА с учётом контекста. Пусть P_i^d это мощность, назначенная компоненту i при условии вождения d . Основные компоненты включают синхронный двигатель с постоянными магнитами (permanent magnet synchronous motor, PMSM), кондиционер, электронный блок управления (electronic control unit, ECU), информационно-развлекательную систему и освещение. Когда $(\widehat{SOC}) \geq 40\%$, оптимальные значения мощности назначаются в зависимости от условия вождения (см. уравнение (3)). Когда SoC снижается ниже порога 40%, система переходит в экономичный режим, постепенно уменьшая мощность (см. уравнение (4)).

$$P_i = \begin{cases} P_i^{highway}, & d = 0 \\ P_i^{urban}, & d = 1 \\ P_i^{downhill}, & d = 2 \end{cases} \quad (3)$$

$$P_i = \lambda \cdot P_i^{initial}, \text{ where } \lambda \in (0,1), \text{ for } 10\% < \widehat{SOC} < 40\% \quad (4)$$

Это пропорциональное масштабирование обеспечивает непрерывность работы при сохранении долговечности аккумулятора. Профиль мощности каждого компонента предварительно охарактеризован и сохранен в таблице поиска, полученной из эмпирических испытаний, что обеспечивает точную настройку при изменяющихся нагрузках.

Когда оценка SoC падает ниже 10%, система ЭА переходит в критический режим выживания. В этом режиме все несущественные подсистемы, такие как кондиционер и информационно-развлекательная система, отключаются. Только критически важные системы – включая ECU, аварийные огни и телеметрию – получают минимальную жизнеспособную мощность P_i^{min} (см. уравнение (5)).

$$P_i = \begin{cases} 0, & i \in \{AC, Infotainment\} \\ P_i^{min}, & i \in \{ECU, Telematics, Lights\} \end{cases} \quad (5)$$

Одновременно активируется модуль определения местоположения автомобиля для поиска ближайшей зарядной станции на основе текущих координат GPS и ограничений по запасу энергии. Затем с помощью алгоритма Дейкстры вычисляется путь до зарядной станции с минимальными затратами энергии. Кроме того, на панели управления появляется сигнал высокого приоритета, информирующий водителя о низком уровне заряда и ближайшей точке подзарядки. Этот проактивный протокол минимизирует риск полного истощения энергии, повышая как безопасность, так и надежность.

4. Аппаратная реализация адаптивного алгоритма управления энергопитанием

Аппаратная реализация предложенного адаптивного алгоритма управления энергопитанием, основанного на Bi-LSTM и алгоритме решающих деревьев, выполнена на платформе Xilinx Zynq UltraScale+ MPSoC (рис. 2). Эта гетерогенная вычислительная система объединяет программируемую логику (PL) и процессорную систему (PS), что делает её особенно подходящей для задач в реальном времени, таких как управление электромобилем (ЭМ) при изменяющихся дорожных условиях. Архитектура проекта построена модульно и включает

пять основных Verilog-модулей: SensorModule, BatteryModule, BiLSTModule, DecisionTreeModule и ActuatorModule, управляемых верхнеуровневым модулем EVPowerManagementTop [14-15].

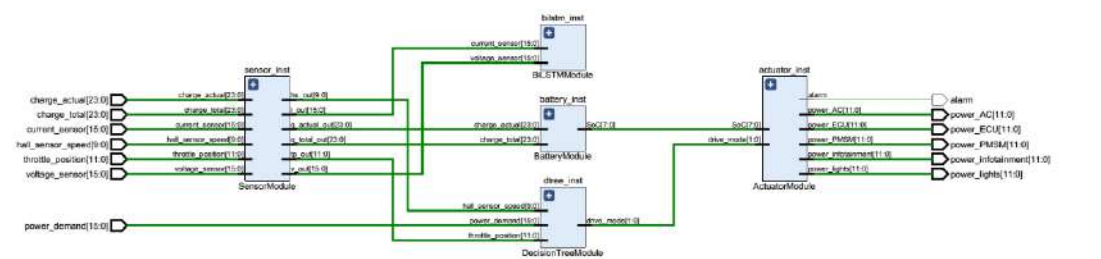


Рис. 2. Реализация предлагаемого адаптивного алгоритма управления мощностью на ПЛИС.
Fig. 2. FPGA implementation of the proposed Adaptive Power Management Algorithm.

SensorModule служит интерфейсом с датчиками автомобиля. Он принимает данные о положении дроссельной заслонки с разрешением 12 бит, данные скорости с датчика Холла – 10 бит, а также значения тока и напряжения по 16 бит. Кроме того, данные о фактическом и общем заряде аккумулятора представлены в виде 24-битных регистров, обеспечивая высокую точность расчёта уровня заряда батареи (SoC).
Модуль BatteryModule вычисляет текущий уровень заряда (SoC) по формуле (6).

$$SoC = \frac{Q_{deliverd}}{Q_{total}} \times 100 \% \tag{6}$$

где $Q_{deliverd}$ и Q_{total} это значения заряда в 24-битном формате. Результат (SoC) представлен 8-битным числом, указывающим процент заряда от 0 до 100%.
BiLSTModule, как правило, реализуется программно, но в данной системе он представлен аппаратным интерфейсом, получающим 16-битные данные о токе и напряжении и прогнозирующим уровень заряда (SoC) на будущее. В реальном применении модель Bi-LSTM может быть обучена, квантована и развернута на AI Engine или PS части Zynq с помощью Xilinx Vitis AI. Выходное значение SoC также составляет 8 бит.
DecisionTreeModule классифицирует дорожные условия (трасса, городской режим, спуск) с помощью логики решающего дерева. Он использует 12-битный вход положения дросселя, 10-битную скорость и 16-битную величину требуемой мощности. На выходе формируется 2-битный код, указывающий режим движения, который используется для выбора режима распределения энергии [11].
ActuatorModule использует входные данные SoC (8 бит) и режим движения (2 бита) для принятия решений по распределению мощности на следующие подсистемы: двигатель PMSM, кондиционер, информационно-развлекательная система, электронный блок управления (ECU) и осветительные фары. Каждому компоненту присваивается выходное значение в 12 бит, что обеспечивает диапазон мощности до 4096 Вт/ч. Также предусмотрен сигнал тревоги (alarm), активируемый при снижении уровня SoC ниже 10% – он инициирует процедуру обнаружения ближайшей зарядной станции по интерфейсу сети контроллеров (controller area network, CAN).
Все выходные данные передаются по интерфейсу CAN, реализованному в PL, что обеспечивает совместимость со стандартами автомобильных сетей. Форматирование и передача кадров CAN реализованы на уровне Verilog-модулей.

5. Результаты и обсуждение

5.1 Генерация и моделирование набора данных с использованием ПЛИС

В данном исследовании используется испытательный стенд на базе ПЛИС (field-programmable gate array, FPGA) для симуляции и генерации набора данных, необходимых для алгоритма адаптивного управления питанием, который интегрирует Bi-LSTM и дерево решений для оптимального распределения энергии в электромобилях. Реализация на ПЛИС моделирует электрические, механические и эксплуатационные характеристики типичного электромобиля среднего класса, и генерирует ключевые входные параметры, такие как положение дросселя, скорость автомобиля, напряжение и ток аккумулятора, а также координаты GPS с заданной периодичностью (например, каждые 30 секунд). Эти параметры используются для симуляции различных условий вождения (город, шоссе, спуск), что отражает реалистичное поведение ЭВ.

Проект на ПЛИС реализован на языке Verilog, где код предназначен для генерации сенсорных данных в пределах заданных рабочих диапазонов. Например, положение дросселя T_p генерируется случайным образом в пределах от 0% до 100%, имитируя действия водителя. Скорость автомобиля V_s вычисляется по формуле, $V_s = \alpha \cdot T_p + \beta$ где α коэффициент масштабирования, а β базовая скорость при нулевом дросселе. Напряжение аккумулятора V_b моделируется как: $V_b = V_{max} - k \cdot (100 - SoC)$ где V_{max} – максимальное напряжение аккумулятора, а k – коэффициент падения напряжения на процент снижения заряда. Ток аккумулятора I_b определяется уравнением мощности: $P = V_b \cdot I_b$, где P требуемая мощность двигателя ЭВ, а V_b напряжение батареи.

Код Verilog, используемый для этой реализации, подключает стандартные библиотеки IEEE.STD_LOGIC_1164 и IEEE.STD_LOGIC_ARITH.ALL, которые обеспечивают логические и арифметические функции. Данные функции используются в испытательном стенде на ПЛИС для генерации реалистичных данных, которые в дальнейшем подаются на вход алгоритму адаптивного управления питанием. Алгоритм на основе Bi-LSTM и дерева решений определяет распределение мощности между различными подсистемами электромобиля (например, двигатель PMSM, кондиционер, телеметрия, ECU).

5.2 Оценка адаптивного блока управления энергопотреблением: использование ресурсов и показатели производительности

Для проверки эффективности и реализуемости предложенного алгоритма для электрических транспортных средств была проведена подробная оценка производительности. Оценка сосредоточена на сравнении предложенного адаптивного блока управления энергопотреблением на базе ПЛИС (adaptive power management unit, APMU), интегрирующего Bi-LSTM и логику дерева решений, с традиционным блоком управления аккумулятором (battery management unit, BMU) по таким параметрам, как эффективность энергопотребления, скорость вычислений и использование аппаратных ресурсов. Оценка включает в себя метрики в реальном времени, полученные из симуляций на платформе Zynq UltraScale+ MPSoC, и данные о потреблении ресурсов, полученные из отчетов синтеза ПЛИС. Табл. 2 предоставляет метрики использования ресурсов ПЛИС для предложенного APMU, реализованного на платформе Zynq UltraScale+ MPSoC. Разбивка включает логические срезы, таблицы поиска (look-up tables, LUT), триггеры (flip-flops, FF), блоки памяти (Block RAM, BRAM) и единицы цифровой обработки сигналов (digital signal processing, DSP), использованные при синтезе и реализации.

Модуль Bi-LSTM занимает основную часть использования DSP из-за операций умножения матриц в вычислениях логических вентилей и переходах состояний. Для этого модуля выделено

примерно 62 среза DSP. Логика дерева решений, напротив, требует минимальных арифметических операций и реализована с использованием небольшого количества LUT и FF. Общее использование срезов составляет менее 28% от доступных ресурсов, что подтверждает масштабируемость проекта для добавления дополнительных модулей или расширения глубины сети. Кроме того, проект занимает менее 15% BRAM, что обеспечивает эффективное использование памяти на чипе. Такой небольшой объем используемых ресурсов позволяет достичь производительности в реальном времени без ущерба для вычислительной сложности.

Табл. 2. Использование ресурсов ПЛИС для предложенного APMU.
Table 2. FPGA Resource Utilization for the Proposed APMU.

Модуль	Функция	Площадь кристалла (мкм ²)	Оценочное количество вентиляей
Блок обработки Bi-LSTM	Оценка уровня заряда батареи (SoC) на основе данных сенсоров	58000	145000
Классификатор на основе решающего дерева	Классификация условий вождения	15200	42000
Генератор положения дросселя	Симуляция пользовательского входа (0–100%)	3200	8000
Блок моделирования скорости	Расчёт скорости (V_s) на основе положения дросселя	2400	6000
Оценка напряжения батареи	Моделирование напряжения (V_b) в зависимости от SoC	2800	6800
Блок расчёта тока батареи	Расчёт I_b на основе V_b и потребляемой мощности	3600	8500
Генератор GPS-координат	Генерация долготы и широты в пределах региона эксплуатации	4000	9000
Управляющий блок (конечный автомат)	Контроль за потоками данных и последовательностью операций	6800	17500
Модуль интерфейса памяти	Обмен данными с блоками BRAM	8200	20000
Контроллер распределения мощности	Назначение мощности по устройствам в зависимости от условий	7600	18000

Синтез был выполнен с использованием Vivado 2023.2 с IP-ядрами от Xilinx Deep Learning Processor Unit (DPU) и пользовательскими RTL-модулями, написанными на Verilog. Анализ времени подтверждает, что проект соответствует временным ограничениям при частоте тактового сигнала 100 МГц, что дополнительно подтверждает его пригодность для встроенных автомобильных приложений. Табл. 3 представляет собой комплексную сравнительную оценку предложенного адаптивного блока управления энергопотреблением (APMU), реализованного на платформе ПЛИС с логикой Bi-LSTM и дерева решений, и традиционного блока управления аккумулятором (BMU), используемого в стандартных электрических транспортных средствах среднего класса. Включенные параметры являются показателями производительности системы, эффективности аппаратных ресурсов и отклика в реальном времени. Ключевые метрики, такие как потребление энергии, задержка вычислений и пропускная способность, представлены для подчеркивания операционной превосходства предложенного решения на базе ПЛИС. В частности, APMU потребляет всего 0,98 Вт, что почти на 50% меньше, чем традиционный BMU. Это преимущество в энергопотреблении объясняется эффективным синтезом логики, затвором тактов и оптимизированным доступом к памяти. Что касается скорости вычислений, то APMU достигает задержки принятия решений всего 5,6 мкс, по сравнению с 150–400 мкс в традиционных BMU, что связано с аппаратной параллельностью и функциями

конвейеризации. Полученная пропускная способность составляет около 178 571 операций в секунду, что значительно превышает возможности систем на базе микроконтроллеров.

Табл. 3. Сравнительная оценка предложенного APMU и традиционного BMU.

Table 3. Comparative Evaluation of the Proposed APMU vs. Conventional BMU.

Параметр	Предлагаемый APMU (на базе ПЛИС)	Традиционный BMU
Потребляемая мощность	≈ 0.98 Вт	$\approx 1.8 - 2.5$ Вт
Задержка вычислений	≈ 5.6 мкс	$\approx 150 - 400$ мкс
Пропускная способность	$\approx 178,571$ операций/сек	$\approx 2,500 - 6,700$ операций/сек
Используемая память (BRAM / RAM)	≈ 43 Кб BRAM	$\approx 128 - 512$ Кб SRAM/EEPROM
Энергоэффективность	≈ 181.6 операций/Вт	$\approx 3 - 10$ операций/Вт
Архитектура обработки	Bi-LSTM + дерево решений (адаптивная)	Правило-ориентированная или таблицы поиска
Аппаратная платформа	Zynq UltraScale+ MPSoC	MCU / DSP на базе микроконтроллера
Адаптивность к условиям	Высокая (модель переобучаема)	Низкая (нужна прошивка)

Энергетическая эффективность, определяемая как операции на ватт, также значительно выше в предложенном подходе благодаря оптимизации модели глубокого обучения и аппаратному ускорению. Кроме того, архитектура на основе Bi-LSTM и дерева решений позволяет обеспечивать адаптивность в реальном времени, что ограничено в традиционных правилах BMU.

5.3 Сравнительный анализ методов оценки уровня заряда

Анализ двух визуализаций – «Уровень заряда (SoC, %) с течением времени» и «Тепловая карта показателей ошибок» – позволяет получить важные сведения о точности различных методов оценки SoC в условиях городской езды (рис. 3). Городская среда характеризуется частыми остановками и стартами, переменной скоростью движения, простым на светофорах и применением рекуперативного торможения. Эти особенности создают нелинейные и трудно предсказуемые шаблоны разряда аккумулятора, что делает точную оценку SoC особенно критичной для эффективности и безопасности электромобилей.

На линейном графике, озаглавленном «Уровень заряда (SoC, %) с течением времени для различных методов оценки», сравниваются пять методов: H-infinity фильтр, фильтр Калмана (KF), метод учёта заряда (Charge Counting), LSTM и Bi-LSTM (двунаправленная LSTM-сеть). Начальное значение SoC составляет около 70% для всех методов, с постепенным снижением по мере времени. Метод H-infinity демонстрирует более консервативную оценку SoC, что отражает его устойчивый подход, направленный на надёжность в условиях неопределённости. В то же время, KF постоянно переоценивает SoC, что связано с его зависимостью от линейных моделей, которые не всегда применимы в динамичных условиях городской среды. Метод учёта заряда прост в реализации, но подвержен накоплению ошибок интегрирования, особенно при часто изменяющихся нагрузках и рекуперации.

LSTM и Bi-LSTM превосходят традиционные методы за счёт способности эффективно улавливать временные зависимости и сложные нелинейные закономерности в данных. Bi-LSTM, обучаясь в обоих временных направлениях, может предсказывать поведение SoC с наибольшей точностью, особенно в условиях резких изменений нагрузки и скорости движения в городе. Это делает Bi-LSTM особенно пригодным для применения в интеллектуальных системах управления аккумуляторами электромобилей.

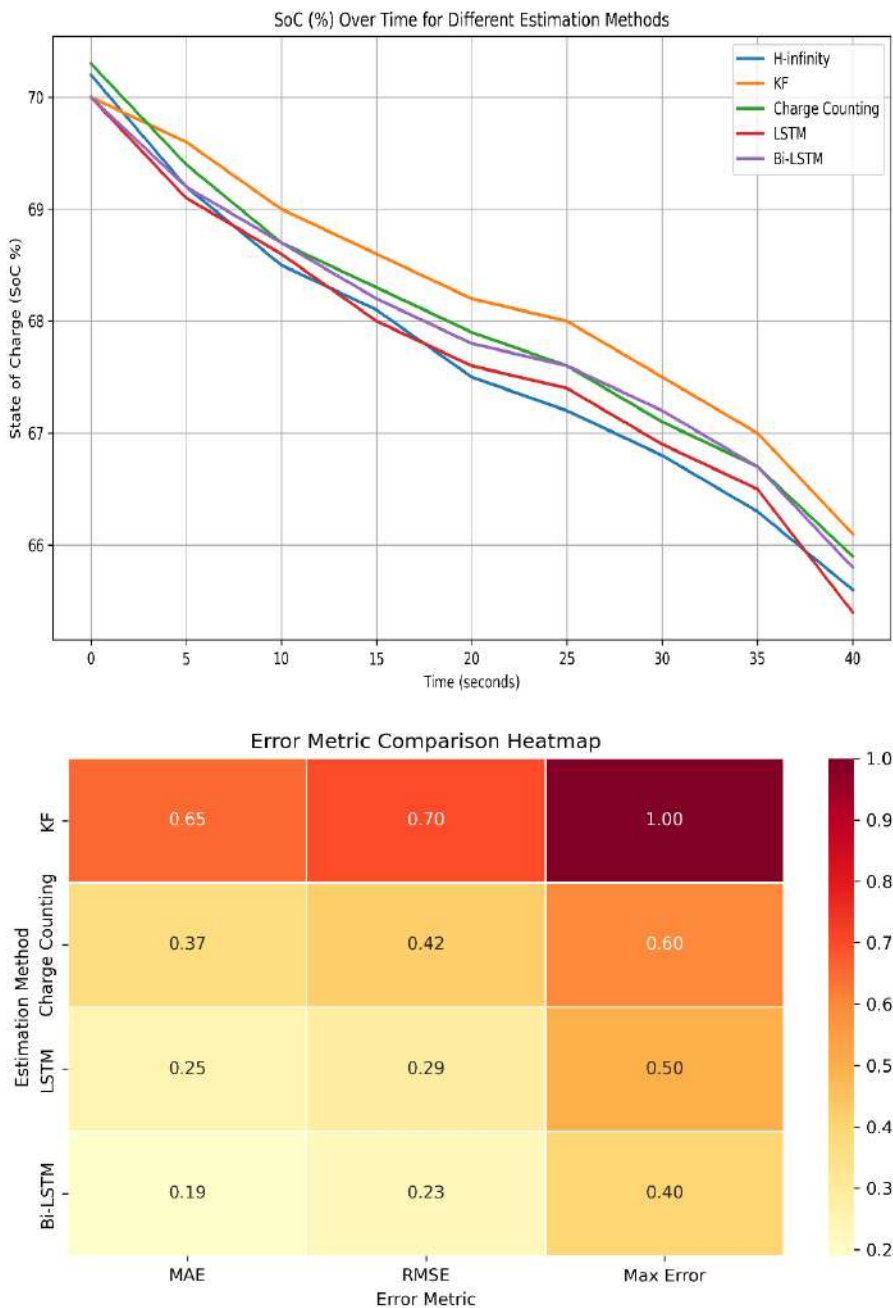


Рис. 3. Сравнительный анализ методов оценки уровня заряда аккумулятора:
сверху – процент уровня заряда (SoC) во времени для различных методов оценки;
ниже – средняя абсолютная ошибка (MAE), среднеквадратичная ошибка (RMSE) и максимальная
ошибка для различных методов оценки относительно эталонного метода H-infinity.

Fig. 3. Comparative Analysis of SoC Estimation Methods:
(upper) State of Charge (SoC) percentage over time for various estimation techniques;
(below) Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Maximum Error
across different estimation methods with respect to the H-infinity reference.

На тепловой карте «Сравнительная тепловая карта метрик ошибок» представлены три показателя оценки точности: средняя абсолютная ошибка (MAE), среднеквадратичная ошибка (RMSE) и максимальная ошибка. Метод KF демонстрирует наибольшие значения ошибок (MAE: 0.65%, RMSE: 0.70%, Max Error: 1.00%), что подчёркивает его ограниченность в сложных условиях. Метод учёта заряда показывает лучшие, но всё ещё нестабильные результаты. LSTM значительно снижает ошибки (MAE: 0.25%, RMSE: 0.29%, Max Error: 0.50%) благодаря глубокому обучению и обработке временных данных. Bi-LSTM достигает наименьших показателей ошибок (MAE: 0.19%, RMSE: 0.23%, Max Error: 0.40%), подтверждая свою адаптивность и точность. Таким образом, визуальный анализ подтверждает, что модель Bi-LSTM превосходит традиционные и другие нейросетевые методы в оценке SoC в условиях городской езды. Благодаря способности учитывать как прошлые, так и будущие значения временных рядов, она обеспечивает высокую точность прогноза, делая её предпочтительным выбором для современных систем управления аккумулятором в электромотоциклах.

5.4 Потребление энергии в электромотоциклах при оптимизированных условиях вождения (SoC: 40% < SoC > 10%)

Потребление энергии различными электрическими компонентами среднего электромотоцикла (EV) было проанализировано в различных условиях вождения (шоссе, город, спуск), с фокусом на оптимизацию использования энергии с помощью предсказательных моделей, таких как Long Short-Term Memory (LSTM) и Bi-LSTM + Decision Tree (DT) (табл. 4). Кондиционер, который потребляет много энергии, обычно потребляет 3000 Вт в стандартных условиях, однако его потребление значительно снижается в оптимизированных сценариях, особенно при движении по шоссе (2400 Вт) и в городских условиях (2200 Вт). Это демонстрирует, что система может быть настроена для более эффективной работы в зависимости от условий вождения, что улучшает энергоэффективность. Аналогичным образом, потребление энергии галогенных ламп высокой интенсивности (HID), хотя и невелико – 35 Вт по умолчанию, снижается немного во время движения по шоссе и при сплошном движении, что отражает более низкие требования к освещению в этих условиях. Система мультимедиа, которая обычно потребляет 600 Вт, также показывает значительное снижение потребления энергии, особенно в городских условиях (480 Вт), что указывает на то, что система адаптируется в зависимости от взаимодействия пользователя и условий вождения.

Электронный блок управления (ECU), потребляющий 600 Вт по умолчанию, также выигрывает от оптимизации, с уменьшением потребления энергии до 420 Вт в городских условиях. Система управления аккумулятором (BMS), отвечающая за мониторинг состояния аккумулятора, потребляет 200 Вт в стандартных условиях, но снижается до 160 Вт при сплошном движении, что также демонстрирует способность системы адаптироваться к условиям эксплуатации. Система рекуперации энергии при торможении, которая восстанавливает энергию при торможении, обычно потребляет 300 Вт, но это значение снижается до 230 Вт при сплошном движении, поскольку система оптимизирует восстановление энергии. Электрический усилитель руля, потребляющий 150 Вт, снижается до 110 Вт во время сплошного движения, что отражает меньшие требования к энергии для рулевого управления при движении накатом. Результаты показывают, что интеграция моделей LSTM и Bi-LSTM + DT значительно оптимизирует потребление энергии различных компонентов автомобиля, что способствует увеличению дальности поездки и улучшению общей эффективности, делая электромотоциклы более устойчивыми в различных условиях вождения. Эта стратегия оптимизации имеет решающее значение для улучшения энергоэффективности и эксплуатационной устойчивости будущих конструкций электромотоциклов.

Табл. 4. Потребление энергии электрических компонентов в электромобилях среднего класса (SoC: 40% < SoC > 10%) при различных условиях вождения.
Table 4. Power Consumption of Electrical Components in Mid-Range Electric Vehicles (10% < SoC < 40%) under Different Driving Conditions.

Электрические компоненты	По умолчанию	Шоссе (LSTM)	Шоссе (Bi-LSTM + DT)	Город (LSTM)	Город (Bi-LSTM + DT)	Спуск (LSTM)	Спуск (Bi-LSTM + DT)
Кондиционер	3000	2700	2400	2600	2200	2500	2000
Освещение (HID лампы и др.)	35	30	28	32	30	30	28
Мультимедиа	600	550	500	520	480	500	450
Электронный блок управления (ECU)	600	500	450	480	420	450	400
Система управления аккумулятором (BMS)	200	180	170	190	180	180	160
Система рекуперации энергии при торможении	300	280	260	270	250	270	230
Электрический усилитель руля	150	130	120	140	130	140	110

5.5 Оценка дальности хода электрического транспортного средства при различных условиях вождения

Оценка дальности хода электрического транспортного средства (ЭТС) при различных условиях вождения проводилась с использованием формулы, основанной на потреблении энергии, где дальность хода для каждого условия была рассчитана с помощью формулы (7). Здесь $Range_{default}$ представляет стандартную дальность хода 200 км при стандартных условиях вождения, а $Total Power_{Default}$ составляет 4885 Вт – общее потребление энергии в этих условиях. Потребление энергии для каждого условия вождения, изменяющееся в зависимости от оптимизационных стратегий, таких как LSTM и Bi-LSTM + DT, использовалось для вычисления нового диапазона ($Total Power_{New}$). Табл. 5 суммирует общее потребление энергии и рассчитанную дальность хода для каждого условия вождения:

$$Range_{New} = Range_{default} \times \frac{Total Power_{Default}}{Total Power_{New}} \times 100 \% \tag{7}$$

Результаты показали, что оптимизационные стратегии привели к значительному улучшению дальности хода ЭТС, причем наибольшее увеличение наблюдается при использовании Bi-LSTM + DT для шоссежного вождения, что дало увеличение дальности на 25%, достигая 250 км. Другие условия, такие как городское вождение и спуск с гор, также показали заметные улучшения. Дальность хода для городского вождения увеличилась на 21,22% при использовании стратегии Bi-LSTM + DT, в то время как для спуска с гор улучшение составило скромные 11,58%. Процентное увеличение для каждого условия вождения было вычислено, и результаты показали, как оптимизационные стратегии LSTM и Bi-LSTM + DT повышают энергоэффективность, особенно в условиях высокой потребности в энергии, таких как шоссе и городской трафик. Эти оптимизации адаптируют потребление энергии транспортным средством в реальном времени, что приводит к увеличению дальности хода, как это показано на круговой диаграмме (рис. 4). В целом, результаты демонстрируют, что стратегии на основе машинного обучения могут эффективно улучшить работу ЭТС, обеспечивая более дальние поездки и улучшенную устойчивость в реальных условиях вождения.

Табл. 5. Расчет дальности хода при различных условиях вождения с оптимизированным потреблением энергии.

Table 5. Range Calculation under Different Driving Conditions with Optimized Power Consumption.

Условия вождения	Общее потребление энергии (Вт)	Рассчитанная дальность хода (км)
По умолчанию	4885	200
Шоссе (LSTM)	4410	221,54
Шоссе (Bi-LSTM + DT)	3908	250,00
Город (LSTM)	4512	216,53
Город (Bi-LSTM + DT)	4030	242,43
Спуск с гор (LSTM)	4870	200,62
Спуск с гор (Bi-LSTM + DT)	4378	223,16

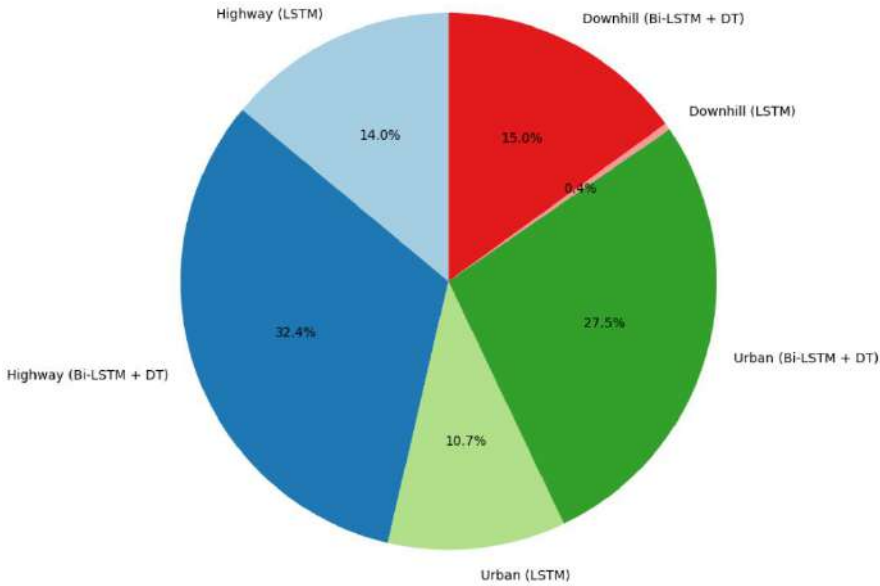


Рис. 4. Круговая диаграмма, иллюстрирующая процентное увеличение дальности хода ЭТС при различных условиях вождения с оптимизированными стратегиями потребления энергии (шоссе, город и спуск с гор) с использованием LSTM и Bi-LSTM + DT.

Fig. 4. Pie chart illustrating the percentage increase in EV range under different driving conditions with optimized power consumption strategies (Highway, Urban, and Downhill) using LSTM and Bi-LSTM + DT.

6. Заключение

В данной работе разработан и реализован адаптивный блок управления питанием (APMU) на базе ПЛИС с использованием платформы Zynq UltraScale+ MPSoC, направленный на повышение энергоэффективности и реализацию интеллектуального распределения мощности в реальном времени для электромобилей. Комбинация Bi-LSTM для оценки состояния заряда и дерева решений для классификации условий движения обеспечивает динамическое управление энергопотреблением на основе реальных параметров. Аппаратная реализация показала существенные преимущества по задержке, пропускной способности, энергопотреблению и использованию ресурсов по сравнению с традиционными BMU. Предложенный подход обеспечил увеличение запаса хода до 25% на шоссе и более чем на 20% в городских условиях при минимальном энергопотреблении и высокой скорости обработки. Полученные результаты подтверждают потенциал ПЛИС-архитектур с применением методов машинного обучения для высокоэффективного управления

энергопитанием в электромобилях нового поколения. В будущем планируется развертывание модели в системах HIL (Hardware-in-the-Loop) и расширение на многомашинные сети с совместным использованием энергии.

Список литературы / References

- [1]. Khan, M. K., Houran, M. A., Kauhaniemi, K., Zafar, M. H., Mansoor, M., & Rashid, S. (2024). Efficient state of charge estimation of lithium-ion batteries in electric vehicles using evolutionary intelligence-assisted GLA-CNN-Bi-LSTM deep learning model. *Heliyon*, 10(15), e35183. <https://doi.org/10.1016/j.heliyon.2024.e35183>.
- [2]. Rahman, T., Miah, M. S., Karim, T. F., Hossain Lipu, M. S., Fuad, A. M., Islam, Z. U., Ali, M. M. N., Shakib, M. N., Sahrani, S., & Sarker, M. R. (2024). Empowering fuel cell electric vehicles towards sustainable transportation: An analytical assessment, emerging energy management, key issues, and future research opportunities. *World Electric Vehicle Journal*, 15(11), 484. <https://doi.org/10.3390/wevj15110484>.
- [3]. Miao, Y., Gao, Y., Liu, X., Liang, Y., & Liu, L. (2025). Analysis of state-of-charge estimation methods for Li-ion batteries considering wide temperature range. *Energies*, 18(5), 1188. <https://doi.org/10.3390/en18051188>. Thangavel, D. Mohanraj, T. Girijaprasanna, S. Raju, C. Dhanamjayulu and S. M. Muyeen, "A Comprehensive Review on Electric Vehicle: Battery Management System, Charging Station, Traction Motors," in *IEEE Access*, vol. 11, pp. 20994-21019, 2023, doi: 10.1109/ACCESS.2023.3250221.
- [4]. Lu, J., Chen, Z., Yang, Y., & L. V., M. (2018). Online estimation of state of power for lithium-ion batteries in electric vehicles using genetic algorithm. *IEEE Access*, 6, 20868–20880. <https://doi.org/10.1109/ACCESS.2018.2824559>.
- [5]. Kim, M., & So, J. (2023). VLSI design and FPGA implementation of state-of-charge and state-of-health estimation for electric vehicle battery management systems. *Journal of Energy Storage*, 73, 108876. <https://doi.org/10.1016/j.est.2023.108876>.
- [6]. Raj, P. J., Prabhu, V. V., & Premkumar, K. (2021). Fuzzy logic-based battery management system for solar-powered Li-ion battery in electric vehicle applications. *Journal of Circuits, Systems and Computers*, 30(03), 2150043. <https://doi.org/10.1142/S0218126621500432>.
- [7]. Merina, R.D., Ram, R.S. & Micheal, L.C.P. (2025) FPGA implementation of intelligent battery management unit (i-BMU) based on dynamic power distribution technique for electric vehicles. *Analog Integr Circ Sig Process* 123, 42. <https://doi.org/10.1007/s10470-025-02370-8>.
- [8]. Xue, F., Ling, Z., Yang, Y., & Miao, X. (2017). Design and implementation of novel smart battery management system for FPGA-based portable electronic devices. *Energies*, 10(3), 264. <https://doi.org/10.3390/en10030264>.
- [9]. Al-Dhaifallah, M., Ali, Z. M., Alanazi, M., & others. (2021). An efficient short-term energy management system for a microgrid with renewable power generation and electric vehicles. *Neural Computing and Applications*, 33, 16095–16111. <https://doi.org/10.1007/s00521-021-06247-5>.
- [10]. Ping, F., Miao, X., Yu, H., & Xun, Z. (2023). An improved LSTNet approach for state-of-health estimation of automotive lithium-ion battery. *Electronics*, 12, 2647. <https://doi.org/10.3390/electronics12122647>.
- [11]. R. S. Ram and M. L. C. Prabhaker, (2023) "Intelligent optimization approaches for a secured dynamic partial reconfigurable architecture-based health monitoring system", *J. Circuits Syst. Comput.*, vol. 32, no. 3. <https://doi.org/10.1142/S0218126623500470>.
- [12]. Merina, R.D., Ram, R.S. & Micheal, L.C.P. (2025) FPGA implementation of intelligent battery management unit (i-BMU) based on dynamic power distribution technique for electric vehicles. *Analog Integr Circ Sig Process* 123, 42. <https://doi.org/10.1007/s10470-025-02370-8>.
- [13]. Saday, A., & others. (2023). FPGA-based battery management system for real-time monitoring and instantaneous SOC prediction. *International Journal of Applied Methods in Electronics and Computers*, 11(1), 55–61. <https://doi.org/10.18100/ijamec.1233451>.
- [14]. Kumar, B., Khare, N., & Chaturvedi, P. K. (2018). FPGA-based design of advanced BMS implementing SoC/SoH estimators. *Microelectronics Reliability*, 84, 66–74. <https://doi.org/10.1016/j.microrel.2018.03.015>.
- [15]. Gatica, G., Ahumada, G., Escobar, J. W., & Linfati, R. (2018). Efficient heuristic algorithms for location of charging stations in electric vehicle routing problems. *Studies in Informatics and Control*, 27(1), 73–82. <https://doi.org/10.24846/v27i1y201808>.

Информация об авторах / Information about authors

Ратинараджан Дейзи МЕРИНА в 2011 году получила степень бакалавра, а в 2022 году магистерскую степень в области компьютерных наук и инженерии. Соискатель степени PhD в области компьютерных наук и инженерии, обладает 6-летним профессиональным опытом выполнения научных исследований. Область научных интересов: кибербезопасность, глубокое обучение и архитектура ПЛИС.

Rathinarajan Daisy MERINA – PhD candidate in Computer Science and Engineering, currently working in academia with 6 years of professional experience. Holds an M.E. in Computer Science and Engineering (2022) and a B.E. in Computer Science and Engineering (2011). Research interests: cybersecurity, deep learning, and FPGA architecture.

Радхакришнан Саравана РАМ – обладатель степени PhD в области информации и связи, доцент факультета электроники и технической связи в филиале Университета Анны, Мадурай, Тамилнад, Индия. Научные интересы: реконфигурируемая архитектура, процессоры, СБИС, беспроводные сенсорные сети, сетевая безопасность и интернет вещей.

Radhakrishnan Saravana RAM – PhD in Information and Communication Engineering, Assistant Professor (Selection Grade) at the Department of Electronics and Communication Engineering, Anna University Regional Campus, Madurai, Tamil Nadu, India. Research interests: reconfigurable architecture, processors, VLSI, wireless sensor networks, network security, and IoT.

Лордвин Сесил Прабхакер МАЙКЛ – обладатель степени PhD в области информационных и коммуникационных технологий (Университет Анны, 2018 г.), профессор кафедры электроники и коммуникационных технологий в Вел Тех МультиТех Инженерный колледж докторов Сакунтхала и Рангараджана, Ченнай, Индия. Область научных интересов: встраиваемые системы, интернет вещей, системы реального времени, интеллектуальный системный дизайн, системы высокопроизводительных вычислений, многоядерные архитектуры и машинное обучение для управления батареями электромобилей. Член научных сообществ IEEE, IAEng, CSTA и HiPEAC, рецензент международных журналов и конференций.

Lordwin Cecil Prabhaker MICHEAL – PhD in Information and Communication Engineering (Anna University, 2018), Professor in the Department of Electronics and Communication Engineering at Vel Tech Rangarajan Dr. Sagunthala R&D Institute of Science and Technology, India. Research interests: embedded systems, IoT, real-time systems, smart system design, HPC systems, multicore architectures, and machine learning for EV battery management. Member of IEEE, IAEng, CSTA, and HiPEAC; reviewer for international journals and conferences.



Применение компьютерного зрения для локализации позвонков на срединных сагиттальных срезах компьютерной томографии

А.Р. Теплякова, ORCID: 0000-0001-8612-3850 <anastasija-t23@mail.ru>

Р.В. Шершнева, ORCID: 0009-0000-2324-5893 <rvshershnev@yandex.ru>

*Обнинский институт атомной энергетики — филиал федерального государственного автономного образовательного учреждения высшего образования «Национальный исследовательский ядерный университет «МИФИ» (ИАТЭ НИЯУ МИФИ),
Россия, 249039, г. Обнинск, тер. Студгородок, д. 1.*

Аннотация. Автоматизация рутинных операций, связанных с анализом медицинских снимков, представляет собой важную задачу, так как позволяет снизить нагрузку на рентгенологов. Выбор снимков компьютерной томографии, соответствующих уровням конкретных позвонков, для оценки состава тела пациента обычно осуществляется вручную, что требует дополнительных временных затрат. Цель исследования – разработка подхода к решению задачи локализации позвонков на срединных сагиттальных срезах компьютерной томографии для автоматического выбора аксиальных срезов, используемых для оценки состава тела. В основе разработанного подхода лежит применение модели мультиклассовой сегментации, имеющей архитектуру семейства U-Net, и методов компьютерного зрения для предобработки снимков и постобработки масок сегментации. С целью оценки влияния типов входных данных и архитектур моделей на точность сегментации рассмотрено 20 конфигураций подхода. Выявлено, что предлагаемый метод предобработки входных данных, основанный на формировании трёхканальных снимков, повышает точность мультиклассовой сегментации для четырёх архитектур из пяти рассмотренных (Dense U-Net демонстрирует максимальное значение коэффициента схожести Дайса – 0,8858). Также установлено, что предложенный метод расширения обучающей выборки, основанный на пропуске аксиальных срезов при формировании сагиттальных срезов, повышает точность мультиклассовой сегментации для моделей с архитектурами ResU-Net и Dense U-Net. На базе предложенного подхода реализован программный модуль, решающий задачи автоматического определения положений позвонков шейного, грудного и поясничного отделов позвоночника на срединном сагиттальном срезе компьютерной томографии, их визуализации и определения индексов аксиальных срезов, соответствующих центрам тел позвонков. Разработанный модуль интегрирован с программой для визуализации и анализа медицинских файлов DICOM. Разработанный модуль может применяться в качестве вспомогательного инструмента при решении диагностических задач.

Ключевые слова: компьютерное зрение; глубокое обучение; компьютерная томография; сегментация; локализация; предобработка; аугментация; позвонок; архитектура U-Net; медицинские снимки.

Для цитирования: Теплякова А.Р., Шершнева Р.В. Применение компьютерного зрения для локализации позвонков на срединных сагиттальных срезах компьютерной томографии. Труды ИСП РАН, том 37, вып. 4, часть 1, 2025 г., стр. 233–248. DOI: 10.15514/ISPRAS-2025-37(4)-14.

Application of Computer Vision for Localization of Vertebrae on Midsagittal Computed Tomography Slices

A.R. Teplyakova, ORCID: 0000-0001-8612-3850 <anastasija-t23@mail.ru>

R.V. Shershnev, ORCID: 0009-0000-2324-5893 <rvshershnev@yandex.ru>

*Obninsk Institute for Nuclear Power Engineering,
1, Studgorodok territory, Obninsk, 249039, Russia.*

Abstract. Automation of routine operations related to medical image analysis is an important task, as it reduces the workload of radiologists. The selection of computed tomography images corresponding to the levels of specific vertebrae for assessing the patient's body composition is usually done manually, which requires additional time. The purpose is to develop an approach to solving the problem of vertebrae localization on midsagittal computed tomography slices for automatic selection of axial slices used to assess body composition. We developed an approach based on the use of a multiclass segmentation model with the U-Net family architecture and computer vision methods for images preprocessing and segmentation masks postprocessing. In order to assess the impact of input data types and model architectures on segmentation accuracy, we considered 20 approach configurations. We found that the proposed method of preprocessing input data, based on the formation of three-channel images, increases the accuracy of multiclass segmentation for four architectures out of five considered (Dense U-Net demonstrates the maximum Dice similarity coefficient of 0.8858). We also found that the proposed training set augmentation method based on skipping axial slices when forming sagittal slices improves the multiclass segmentation accuracy for models with the ResU-Net and Dense U-Net architectures. Based on the proposed approach, we implemented a software module that solves the problems of automatic determination of the positions of the cervical, thoracic and lumbar vertebrae on the midsagittal computed tomography slice, their visualization and determination of the axial slice indices corresponding to the vertebral body centers. We integrated the developed module with the program for visualization and analysis of DICOM medical files. The developed module can be used as an auxiliary tool in solving diagnostic problems.

Keywords: computer vision; deep learning; computed tomography; segmentation; localization; preprocessing; augmentation; vertebra; U-Net; medical images.

For citation: Teplyakova A.R., Shershnev R.V. Application of Computer Vision for Localization of Vertebrae on Midsagittal Computed Tomography Slices. *Trudy ISP RAN/Proc. ISP RAS*, vol. 37, issue 4, part 1, 2025, pp. 233-248 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-14.

1. Введение

Системы поддержки принятия врачебных решений, в основе функционирования которых лежит применение методов компьютерного зрения, способны выступать в качестве источника второго мнения и снижать рутинную нагрузку на рентгенологов при анализе снимков, получаемых методами медицинской визуализации. Одним из этапов решения диагностических задач, связанных с оценкой состава тела по снимкам компьютерной томографии (КТ), является выбор снимков, соответствующих определённым позвонкам. Для оценки скелетно-мышечного индекса (СМИ) чаще всего используется срез, соответствующий центру тела позвонка L3. Однако снимки КТ уровня L3 не всегда доступны. В связи с этим авторы [1] исследуют, в какой степени снимки КТ на уровне других позвонков могут быть полезны для оценки состояния мышечной ткани, и делают вывод, что для СМИ значительны корреляции между уровнями C1–C2, C7–T5, T7–L5 и L3 ($r=0,70-0,93$) соответственно. Авторы подчеркивают, что при отсутствии срезов на уровне позвонка L3 срезы на уровне позвонков грудного и поясничного отделов (T1–T5, T7–L5) можно использовать для оценки количественных и качественных показателей мышечной ткани у пациентов с раком головы и шеи, пищевода, легких и меланомой, однако использование срезов на уровне позвонков шейного отдела (C1–C2, C7) обеспечивает более низкую надежность. Авторы [2] показывают, что оценки СМИ, получаемые с использованием

площади мышечной ткани среза КТ на уровне T4, сопоставимы с оценками на базе измерений на уровне L3 у пациентов с раком прямой кишки. В статье [3] подчёркивается, что в случаях проведения КТ грудной клетки снимки уровня позвонка L3 могут быть недоступны. В связи с этим авторы исследования демонстрируют, что для проведения оценки состава тела у пациентов с раком лёгкого также возможно использование снимка КТ на уровне позвонка L1. Авторы [4] оценивают корреляции индексов массы тела, полученных через измерение для всего тела и через оценку по одному или нескольким срезам КТ, и приходят к выводу о том, что усреднение по нескольким срезам на уровнях L1–L3 при вычислении оценок может повысить значение корреляции.

Для упрощения одного из этапов анализа снимков КТ с целью оценки состава тела актуальной является разработка подхода к решению задачи локализации позвонков шейного, грудного и поясничного отделов позвоночника, делающее возможной автоматизацию выбора аксиальных срезов КТ, соответствующих уровням определенных позвонков. С учётом того, что авторами настоящей работы ранее обучены модели сегментации мышечной ткани на снимках КТ, соответствующих позвонкам грудного и поясничного отделов [5], подход позволит автоматизировать этап отбора снимков, которые могут быть обработаны этими моделями.

2. Обзор существующих подходов

В рамках исследования был проведён обзор существующих подходов к решению задачи автоматической сегментации позвонков, которую необходимо выполнять для их точной локализации. В подходе, описанном в статье [6], аксиальные срезы КТ делятся на неперекрывающиеся патчи, которые передаются модели на базе многоуровневого разреженного автокодировщика (SSAE). Выполняется операция скользящего окна и для каждого патча определяется, относится ли он к областям позвонков или нет. Авторы [7] предлагают полуавтоматический подход для сегментации позвоночника по данным КТ, основанный на геометрическом потоке с предварительным применением анизотропной диффузной фильтрации. Для сегментации отдельного позвонка пользователю необходимо вручную выбирать исходную точку. В статье [8] предлагается подход на базе двухэтапной Dense-U-Net, последовательно выполняющей локализацию (с помощью 2D-Dense-U-Net) и сегментацию (с использованием 3D-Dense-U-Net) позвонков. Авторами [9] предложена модель, работающая в два этапа: сначала с помощью CHASPPRAU-Net (модификация архитектуры U-Net) сегментируется позвоночник, после чего при помощи MRU-Net (мобильная остаточная U-Net) распознаются отдельные позвонки. В [10] предлагается гибридный подход к сегментации позвонков на аксиальных срезах КТ, сочетающий в себе свёрточную нейронную сеть (CNN) и полностью свёрточную сеть (FCN). Предлагаемая в [11] модель глубокого доверия на основе патчей (PaDBN) автоматически выбирает признаки из патчей изображений КТ, после чего измеряет межклассовые различия. Полученные дискриминативные признаки подаются на вход классификатора, который оценивает вероятность принадлежности к области позвонка. Авторы [12] предлагают метод автоматической сегментации поясничных позвонков, состоящий из двух частей: локализации и сегментации позвонков поясничного отдела. За первый этап отвечает U-Net, за второй – трёхмерная XUnet. CNN для сегментации позвонков Verte-Box, описанная в статье [13], использует механизм контроля для улучшения признаков и подавления информации о помехах, а также включает в себя многомасштабный блок свёртки, обеспечивающий возможность комбинирования размеров рецептивных полей.

Все рассмотренные подходы объединяет то, что они решают задачу трёхмерной сегментации позвонков, подразумевающую необходимость значительных вычислительных затрат для локализации определённого позвонка. В рамках настоящего исследования задача локализации и сегментации позвонков рассматривается как промежуточный этап, необходимый для нахождения срезов, представляющих интерес для дальнейшей обработки.

Одним из требований к реализуемому подходу, который выступает в качестве основы модуля, расширяющего функционал разработанного ранее средства просмотра файлов DICOM [14], является возможность его выполнения как на персональных компьютерах, так и в облачной среде. При первом варианте необходимо учитывать возможные ограничения, накладываемые вычислительным устройством пользователя (например, такие как отсутствие графического процессора). При втором варианте стоит придерживаться подхода, в рамках которого минимизируется объём данных, передаваемых по сети. С учётом этих ограничений в данном исследовании предлагается подход, анализирующий только одно изображение, соответствующее срединному сагиттальному срезу КТ. Преимущество такого подхода в том, что он сочетает в себе высокую точность, малые вычислительные и временные затраты, необходимые для предсказания, а в случае облачного развёртывания позволяет распределять нагрузку между серверной и клиентской частями, кратно снижая сетевой трафик (при формировании срединного сагиттального среза на стороне клиента нет необходимости передачи на сервер полного набора аксиальных срезов исследования КТ). Повышение точности обеспечивается за счёт применения предлагаемых методов обработки снимков. Первый из них основан на формировании трёхканальных изображений, каналы которых представлены снимками в исходном виде, с адаптивным выравниванием гистограмм с ограниченным усилением контраста (CLAHE, Contrast Limited Adaptive Histogram Equalization) и с применением анизотропной диффузной фильтрации (ADF, Anisotropic Diffusion Filtering). Второй метод применяется для расширения обучающей выборки, основан на пропуске аксиальных срезов при формировании изображений, соответствующих сагиттальным срезам.

3. Постановка задачи

При проведении диагностики, которая подразумевает выполнение количественной оценки состава тела человека по отдельным снимкам КТ на уровнях тел определенных позвонков, врачу-рентгенологу сначала необходимо вручную выбирать срезы, соответствующие этим уровням, чтобы дальше выполнять оконтуривание представляющих интерес тканей в ручном режиме или при помощи программного обеспечения для автоматической сегментации. Задача заключается в разработке программного модуля, который извлекает из DICOM файлов со срезами исследования КТ изображение, содержащее срединный сагиттальный срез, после чего выполняет сегментацию позвонков на этом изображении с целью их локализации и определения индексов аксиальных срезов, соответствующих выбираемому пользователем позвонкам. В рамках данной задачи могут быть выделены следующие подзадачи:

- поиск данных для обучения моделей сегментации позвонков и разработки алгоритмов (снимки КТ с мультиклассовыми масками);
- разработка алгоритмов извлечения и предварительной обработки изображений, содержащих срединные сагиттальные срезы КТ;
- построение, обучение и тестирование моделей мультиклассовой сегментации позвонков;
- разработка алгоритмов постобработки масок сегментации (необходимо убрать с них отростки позвонков, оставить только тела) и определения индекса искомого аксиального среза;
- разработка алгоритма визуализации полученных масок позвонков для возможности осуществления пользователем визуального контроля корректности выбора среза;
- интеграция разработанного модуля с программным средством для визуализации и анализа DICOM файлов.

4. Описание предлагаемого подхода

Обработка исследования КТ, состоящего из набора файлов DICOM, в рамках предлагаемого подхода реализуется в три этапа.

4.1 Этап 1: Извлечение и предобработка срединного сагиттального среза

Так как результат КТ представляет собой набор аксиальных срезов, хранящихся в отдельных файлах DICOM, для получения массива пикселей, соответствующего срединному сагиттальному срезу, необходимо сначала осуществить циклический обход аксиальных срезов. Массивы пикселей, соответствующие аксиальным срезам, сортируются по значению элемента данных DICOM, идентифицируемого тегом Slice Location (0020, 1041). Формируется трёхмерный массив пикселей, из которого вырезается двумерный массив, соответствующий срединному сагиттальному срезу. Далее это изображение масштабируется по высоте с использованием коэффициента, рассчитанного на базе значений элементов данных DICOM, идентифицируемых тегами Pixel Spacing (0028, 0030) и Slice Thickness (0018, 0050). В зависимости от протяжённости области тела, охватываемой при проведении КТ, соотношения высоты и ширины срединного сагиттального среза могут значительно варьироваться, поэтому для обработки снимков выбрана стратегия заполнения. Пусть изображение имеет размер $W \times H$, при этом $W < H$. Чтобы получить квадратное изображение (будет подаваться на вход моделей сегментации) с применением стратегии заполнения, необходимо создать массив чёрных пикселей размера $H \times H$, после чего разместить в нём исходное изображение, отступив от левого края $0,5 \cdot (H - W)$ пикселей.

Описанная последовательность действий позволяет получить одноканальные изображения в исходном виде. Если обучение или предсказание выполняется на трёхканальных снимках, то в эту последовательность добавляется ещё один шаг – формирование изображений для трёх каналов. Первое из них – исходное, второе – с применением CLAHE с пределом отсечения 2, третье – обработанное методом шумоподавления на базе ADF. Применение шумоподавления на базе ADF позволяет сохранить значимые для решаемой задачи детали исходного изображения, так как этот метод не размывает края [15].

На рис. 1 показан пример тройки изображений, формирующих снимок в трёхканальном виде. Пример сагиттального среза до и после приведения к квадратной форме показан на рис. 2 (для его формирования использовано КТ исследование из набора данных The Cancer Genome Atlas [16]).

Для повышения точности сегментации также предлагается подход к расширению обучающей выборки, учитывающий свойство КТ, влияющее на формирование сагиттальных срезов. Это свойство обуславливает три возможных варианта взаимного расположения соседних аксиальных срезов: они могут перекрываться, располагаться вплотную друг к другу или на расстоянии (всё зависит от значений толщины среза и расстояния между срезами). Для более устойчивой сегментации сагиттальных срезов, полученных для серий КТ, в которых между аксиальными срезами имеются зазоры, применяется аугментация, суть которой заключается в намеренном пропуске ряда аксиальных срезов при формировании дополнительных изображений, соответствующих сагиттальным срезам. Для более наглядной демонстрации различий между изображениями сагиттальных срезов, сформированными на базе одного и того же КТ исследования без пропуска аксиальных срезов и с применением этого подхода, на рис. 3 показаны области этих изображений, ограниченные одинаковыми координатами.

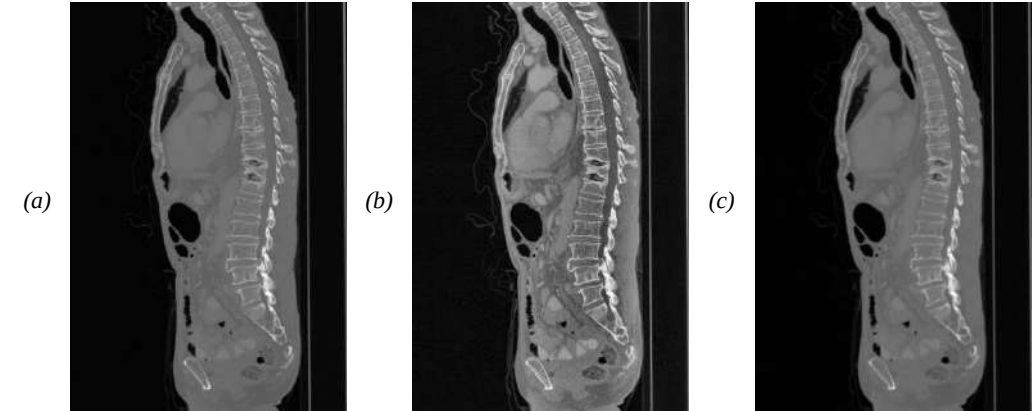


Рис. 1. Составляющие трёхканального изображения:
 снимок в исходном виде (a), с применением CLAHE (b), с применением ADF (c).
 Fig. 1. Components of a three-channel image:
 the image in its original form (a), using CLAHE (b), and using ADF (c).

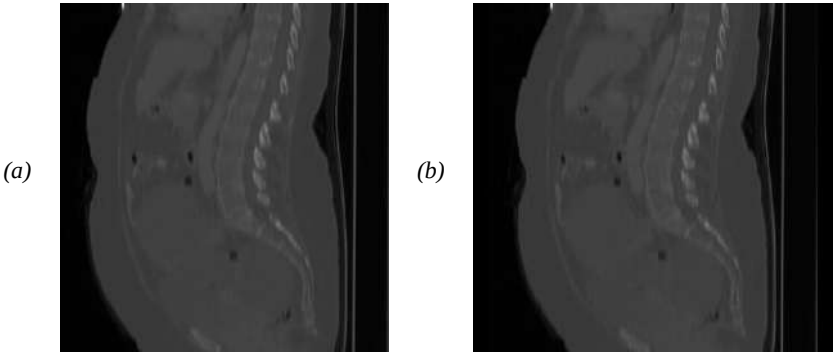


Рис. 2. Срединный сагиттальный срез компьютерной томографии
 в исходном виде (a) и после преобразования к квадратной форме (b).
 Fig. 2. Midsagittal computed tomography image
 in original form (a) and after transformation to a square shape (b).

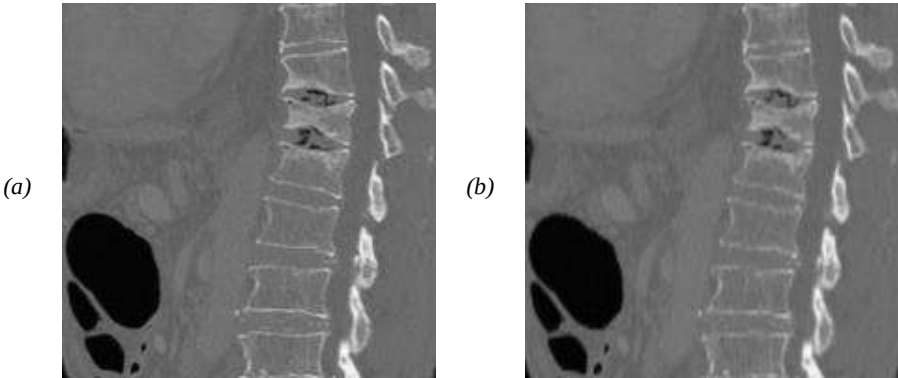


Рис. 3. Сравнение фрагментов изображений срединных сагиттальных срезов, сформированных на
 базе аксиальных срезов одного и того же исследования компьютерной томографии
 без пропуска (a) и с пропуском (b).
 Fig. 3. Comparison of midsagittal slice image fragments generated from axial slices of the same computed
 tomography scan without skipping (a) and with skipping (b).

4.2 Этап 2: Сегментация позвонков на срединном сагиттальном срезе

Полученное на первом этапе изображение, содержащее срединный сагиттальный срез, подаётся на вход обученной модели сегментации, имеющей одну из архитектур U-Net [17], Attention U-Net [18], ResU-Net [19], Inception U-Net [20], Dense U-Net [21]. Предсказание модели представляет собой маску, которая содержит метки классов, соответствующих 24 позвонкам, сегментации которых обучена сеть. В силу того, что маски для обучения моделей взяты из набора для полной сегментации позвонков по трёхмерным массивам пикселей, ненулевыми в масках, выдаваемых моделями, являются пиксели, соответствующие не только телам позвонков, но и отросткам. Однако для решения задачи определения положения тела позвонка необходимо найти аксиальный срез, соответствующий центру тела позвонка, поэтому получаемые маски сначала обрабатываются таким образом, чтобы пиксели, соответствующие отросткам, стали нулевыми (чёрными). Первый шаг алгоритма постобработки заключается в переводе маски в бинарный вид и применении к ней эрозии – морфологической операции, которая основана на вычислении локального минимума для заданного ядра и позволяет сузить яркие области изображения.

Далее производится разбиение маски на прямоугольные сегменты, ширина которых равна ширине самой маски, а высота определяется размерами позвонков путём предварительного анализа размеров ограничивающих их контуров. После этого через оценку положений и размеров контуров, попадающих в выделенные сегменты, производится обнуление тех пикселей, которые относятся к отросткам позвонков. На рис. 4 показана маска в исходном виде и после маскирования отростков на ней.

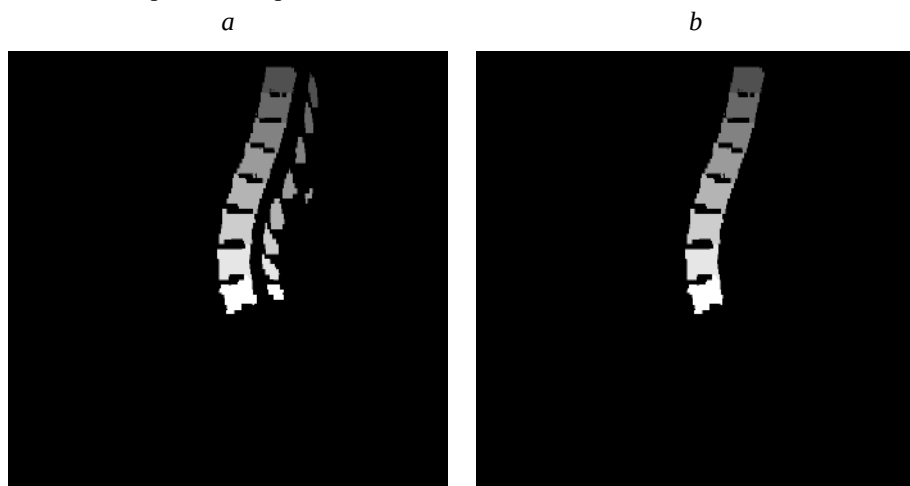


Рис. 4. Маска мультиклассовой сегментации позвонков в исходном виде (a) и после обработки (b).
Fig. 4. Multiclass segmentation mask of vertebrae in the original form (a) and after processing (b).

4.3 Этап 3: Определение положения выбранного позвонка и поиск соответствующего среза

После формирования маски, на которой присутствуют только тела позвонков, выделяется список меток классов позвонков, присутствующих на обработанном срезе. Производится циклический обход списка, при этом на каждой итерации фиксируется положение центра контура, ограничивающего пиксели, которые соответствуют текущей метке. Таким образом, получается список, содержащий метки позвонков и координаты их центров. Далее производится преобразование этих координат с учетом коэффициента, обратного применённому к изображению на этапе 1, и определение индекса аксиального среза,

соответствующего телу центра выбранного позвонка, в массиве. После нахождения нужного аксиального среза формируется изображение, содержащее срединный сагиттальный срез с нанесёнными на него масками, именованиями позвонков, а также горизонтальной линией, соответствующей выбранному аксиальному срезу. Оно может применяться для осуществления визуального контроля корректности выбора среза. Пример такого изображения показан на рис. 5.

Логика функционирования модуля, который разработан на базе описанного подхода, отражена на схеме, показанной на рис. 6.

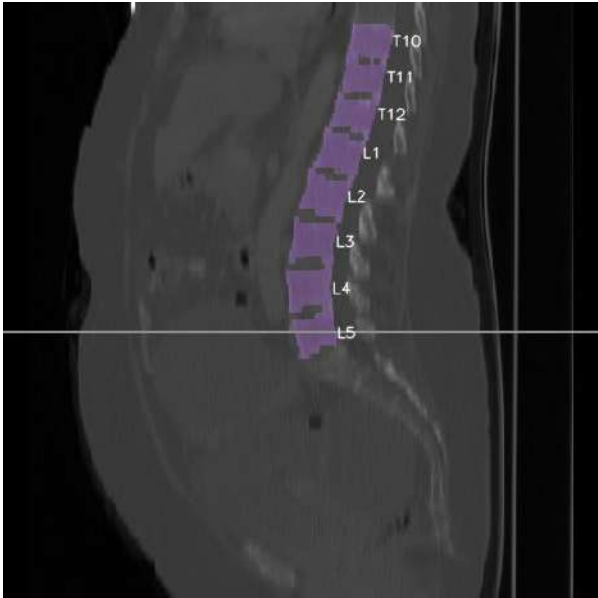


Рис. 5. Пример изображения с нанесенными масками позвонков и линией, соответствующей выбранному аксиальному срезу.
Fig. 5. Example of an image with applied vertebral masks and a line corresponding to the selected axial slice

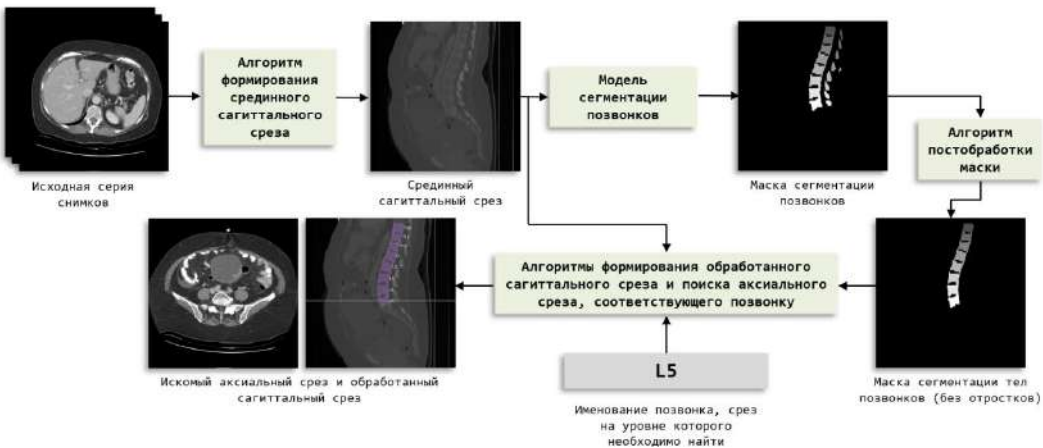


Рис. 6. Логика функционирования разработанного модуля.
Fig. 6. The logic of the functioning of the developed module.

5. Полученные результаты

Программный код разработанного модуля написан на языке Python. В рамках настоящего исследования с использованием библиотеки TensorFlow обучено 20 моделей мультиклассовой сегментации позвонков по срединным сагиттальным срезам КТ, имеющих архитектуры U-Net, Attention U-Net, ResU-Net, Inception U-Net и Dense U-Net. Для каждой из архитектур варьировался подход к формированию обучающих данных, всего их четыре:

- исходные снимки (одноканальный набор);
- трёхканальные снимки (трёхканальный набор);
- исходные снимки с добавлением имитирующих наличие зазора между срезами (одноканальный расширенный набор);
- трёхканальные снимки с добавлением имитирующих наличие зазора между срезами (трёхканальный расширенный набор).

Наборы данных для обучения сформированы на базе набора VerSe [22–24], который содержит 374 исследования 355 пациентов, возраст которых составляет от 42 до 76 лет (использованы данные как VerSe'19, так и VerSe'20). Из каждого исследования выбрано по 15 срезов. Для отбора применялась следующая логика: выбирался срединный сагиттальный срез и по 7 ближайших к нему парасагиттальных срезов с каждой стороны. В тестовую выборку (10% от общего количества срезов) отобраны как срезы в базовом виде, так и срезы с применением подхода к расширению выборки за счёт пропуска аксиальных срезов (исследования, из которых извлекаются срезы, для одноканальных и трехканальных вариантов тестовых снимков идентичны).

Для оценки моделей сегментации применялся коэффициент схожести Дайса (Dice Similarity Coefficient, DSC), для расчёта которого используется следующая формула:

$$DSC = 2 \cdot \frac{|pred \cap ground_truth|}{|pred| + |ground_truth|}$$

где `pred` обозначает маску, предсказанную моделью, а `ground_truth` – истинную маску. В случае мультиклассовой сегментации маски представляют собой одноканальные массивы, значения которых распределены от 0 до 24 (0 соответствует фону, остальные значения – 24 позвонкам). Для расчёта DSC эти маски предварительно преобразуются таким образом, чтобы каждому классу соответствовал отдельный канал, а значения лежали в пределах от 0 до 1 (One Hot Encoding). Полученные значения DSC приведены в табл. 1.

Среди моделей с архитектурой U-Net (самой лёгкой из рассмотренных) максимальное значение DSC (0,7855) для мультиклассовой сегментации продемонстрировала та, которая обучена на трёхканальном наборе без применения предложенного подхода к аугментации. При этом сравнение двух одноканальных моделей с архитектурой U-Net показывает, что расширение выборки за счёт имитации зазоров между аксиальными срезами повышает DSC. Из четырёх моделей с архитектурой Attention U-Net лучшее значение DSC для мультиклассовой сегментации (0,8857) у версии, обученной на трёхканальном наборе без расширения. Заметно, что подход к расширению выборки не оказал положительного влияния на точность мультиклассовой сегментации моделей с архитектурой Attention U-Net при обучении как на одноканальном, так и на трёхканальном наборах. Для моделей с архитектурами ResU-Net и Dense U-Net наиболее точная мультиклассовая сегментация обеспечивается при обучении на данных расширенного трёхканального набора (DSC 0,8162 и 0,8858). Для моделей Inception U-Net лучшее значение DSC мультиклассовой сегментации (0,8127) достигается при обучении на одноканальном наборе без расширения. Для расчёта метрик бинарной сегментации истинные и предсказанные маски мультиклассовой сегментации были предварительно преобразованы в бинарные. Можно заметить, что для всех пяти рассмотренных архитектур наиболее высокие значения DSC в случае бинарной

сегментации достигаются при использовании трёхканальных данных (для Attention U-Net, ResU-Net, Inception U-Net без расширения, U-Net и Dense U-Net – с ним).

Табл. 1. Значения DSC.

Table 1. DSC values.

Типы выходных масок	Архитектуры моделей сегментации				
	U-Net	Attention U-Net	ResU-Net	Inception U-Net	Dense U-Net
Обученные на одноканальном наборе					
мультиклассовые	0,7526	0,8832	0,7879	0,8127	0,8483
бинарные	0,9505	0,9509	0,9386	0,9426	0,9423
Обученные на расширенном одноканальном наборе					
мультиклассовые	0,7588	0,8720	0,7670	0,7183	0,8758
бинарные	0,9570	0,9494	0,9441	0,9404	0,9476
Обученные на трёхканальном наборе					
мультиклассовые	0,7855	0,8857	0,7949	0,7864	0,8503
бинарные	0,9571	0,9553	0,9501	0,9517	0,9492
Обученные на расширенном трёхканальном наборе					
мультиклассовые	0,7224	0,8566	0,8162	0,7478	0,8858
бинарные	0,9602	0,9525	0,9429	0,9412	0,9527

Для проверки того, насколько точно обученные в рамках работы модели способны сегментировать позвонки на срединных сагиттальных срезах исследований не из набора VerSe, дополнительно проведён эксперимент с данными из открытого набора TotalSegmentator [25]. Для этого авторами настоящей работы отобрано 535 исследований КТ из TotalSegmentator, содержащих более 4 позвонков, и выполнены следующие шаги:

1. Из аксиальных снимков исследований КТ сформированы срединные сагиттальные срезы в одноканальном и трёхканальном видах (применялась та же логика, что и для данных VerSe).
2. Из файлов разметки исследований КТ сформированы соответствующие сагиттальным срезам маски позвонков шейного (C1–C7), грудного (Th1–Th12) и поясничного (L1–L5) отделов позвоночника. Позвонкам на полученных масках назначены те же метки, как при работе с VerSe (например, 19 для Th12, 20 для L1 и т. д.). Так как подходы к разметке данных в рассматриваемых наборах имеют различия (для VerSe характерно отсутствие меток для граничных позвонков исследований КТ, тогда как в TotalSegmentator они размечаются), для минимизации их влияния на значения DSC сформированные маски дополнительно обработаны: на них обнулены пиксели, соответствующие граничным позвонкам, которые не сегментируются обученными на данных VerSe моделями.
3. Всеми обученными моделями выполнена сегментация позвонков на сформированных срединных сагиттальных срезах, произведен расчёт значений DSC (приведены в табл. 2).

Максимальные значения DSC мультиклассовой сегментации для четырёх архитектур из пяти рассмотренных достигаются моделями, обученными на трёхканальных данных (для U-Net, ResU-Net, Dense U-Net – с расширением, для Attention U-Net – без расширения). Из четырёх моделей с архитектурой Inception U-Net лучшее значение DSC мультиклассовой сегментации (0,7978) достигается той, которая обучена на одноканальном наборе без расширения. Самые высокие значения DSC мультиклассовой сегментации продемонстрированы моделями с архитектурами Attention U-Net (0,8554) и Dense U-Net (0,8580). Что касается максимальных значений DSC для бинарной сегментации, для трёх архитектур из пяти рассмотренных они достигаются моделями, обученными на трёхканальных данных (для U-Net – с расширением,

для ResU-Net и Dense U-Net – без расширения), а для двух оставшихся (Attention U-Net и Inception U-Net) – моделями, обученными на одноканальном наборе без расширения.

Табл. 2. Значения DSC на данных TotalSegmentator.

Table 2. DSC values on TotalSegmentator data.

Типы выходных масок	Архитектуры моделей сегментации				
	U-Net	Attention U-Net	ResU-Net	Inception U-Net	Dense U-Net
Обученные на одноканальном наборе					
мультиклассовые	0,7487	0,8396	0,7857	0,7978	0,8396
бинарные	0,9120	0,9133	0,9056	0,9065	0,9072
Обученные на расширенном одноканальном наборе					
мультиклассовые	0,7153	0,8412	0,7955	0,7591	0,8248
бинарные	0,9108	0,9120	0,9062	0,9057	0,9082
Обученные на трёхканальном наборе					
мультиклассовые	0,7439	0,8554	0,7732	0,7741	0,8284
бинарные	0,9114	0,9090	0,9097	0,9003	0,9118
Обученные на расширенном трёхканальном наборе					
мультиклассовые	0,7519	0,8272	0,8006	0,7745	0,8580
бинарные	0,9124	0,9069	0,9040	0,9020	0,9083

Оценка временных затрат на мультиклассовую сегментацию позвонков на срединном сагиттальном срезе исследования КТ, реализуемую моделями каждой из рассмотренных архитектур, была проведена на персональном компьютере с 32 ГБ оперативной памяти, центральным процессором (ЦП) Intel Core i7-11800H и графическим процессором (ГП) NVIDIA GeForce RTX 3080 (16 ГБ видеопамати) для двух конфигураций: вычисления на обоих процессорах и только на центральном процессоре. Полученные оценки приведены в табл. 3. Стоит отметить, что всеми обученными моделями обеспечивается малое время предсказания даже в режиме с использованием только ЦП.

Табл. 3. Среднее время сегментации позвонков на срединном сагиттальном срезе, мс.

Table 3. Average time of vertebral segmentation on the midsagittal slice, ms.

Архитектуры моделей сегментации	Конфигурации персонального компьютера	
	ЦП и ГП	ЦП
U-Net	38,7	65
Attention U-Net	52,3	325
ResU-Net	43,2	130
Inception U-Net	57,1	459
Dense U-Net	55,8	226

Что касается экономии, обеспечиваемой при применении для поиска нужного аксиального среза только срединного сагиттального среза, а не полного массива срезов, то она зависит от того, сколько аксиальных срезов содержит исследование и какой подход используется для их анализа. Для примера дополнительно проведён следующий эксперимент: обучена модель бинарной сегментации позвонков с архитектурой Dense U-Net на аксиальных срезах (размер изображения 256×256), реализована сегментация тестовых исследований. Среднее время сегментации отдельного исследования такой моделью составило 22 с при использовании ЦП и ГП (минимальное – 9 с, максимальное – 43 с), в режиме использования только ЦП – 106 с (минимальное – 41 с, максимальное – 216 с). При этом значение DSC, рассчитанное для масок сегментации срединных сагиттальных и ближайших к ним парасагиттальных срезов, извлеченных из полученных трёхмерных массивов масок исследований тестовой выборки, составило 0,9457. Сравнение соизмеримых по сложности моделей сегментации аксиальных

срезов и срединного сагиттального среза показало, что предпочтительным вариантом для решения задачи поиска нужного позвонка выглядит использование моделей, сегментирующих именно сагиттальные срезы.

Модели с архитектурами Dense U-Net и Attention U-Net, обученные на наборах трёхканальных снимков (у Dense U-Net это версия набора с применением предложенного подхода к расширению, у Attention U-Net – без его применения), показали близкие значения DSC для мультиклассовой сегментации, при этом значение DSC для бинарной сегментации выше у Attention U-Net. Это позволяет сделать вывод о том, что обе эти модели могут быть выбраны в качестве лучших из обученных в рамках настоящей работы. С учётом того, что модель с архитектурой Dense U-Net, обученная на наборе трёхканальных снимков с применением предложенного подхода к расширению, обеспечивает более быструю сегментацию при отсутствии ГП, чем модель с архитектурой Attention U-Net, она выбрана для сегментации позвонков на срединном сагиттальном срезе в разработанном модуле (Этап 2). В связи с этим в процедуру формирования данного среза (Этап 1) добавляется шаг, необходимый для преобразования снимка к трёхканальному виду.

Разработанный модуль интегрирован с программой для визуализации и анализа медицинских файлов DICOM. Через интерфейс программы пользователь открывает директорию с файлами DICOM исследования КТ, выбирает из выпадающего списка позвонки, аксиальный срез на уровне которого необходимо найти, после чего нажимает на кнопку «Найти». Модуль выполняет обработку, на интерфейс выводятся искомый аксиальный срез и обработанный срединный сагиттальный срез. Найденный аксиальный срез также может быть обработан сервисом сегментации мышечной ткани, интегрированным с программой визуализации. Модели, применяемые этим сервисом, обучены в программе для обучения моделей сегментации мышечной ткани по снимкам компьютерной томографии [26] и могут сегментировать мышечную ткань на срезах, соответствующих уровням позвонков Th1–Th12, L1–L5 [5]. Отображение результатов поиска нужного среза и его сегментации на интерфейсе программы-просмотрщика показано на рис. 7.

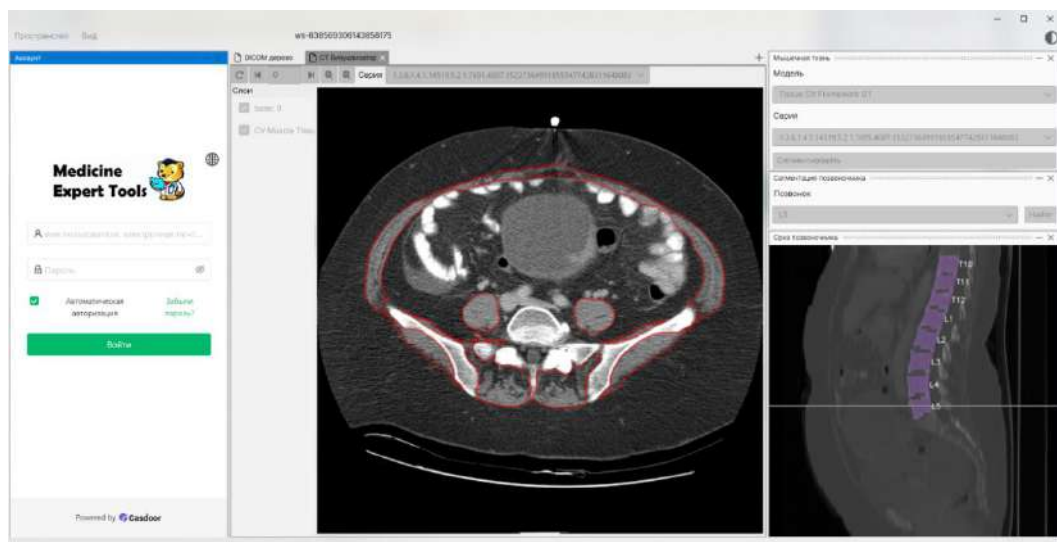


Рис. 7. Пример отображения результатов работы модуля в интерфейсе программы для визуализации и анализа медицинских файлов DICOM.

Fig. 7. An example of displaying the results of the module in the interface of the program for visualization and analysis of DICOM medical files.

6. Заключение

Разработан подход к решению задачи локализации позвонков на срединных сагиттальных срезах компьютерной томографии. На базе разработанного подхода реализован программный модуль, который анализирует серию снимков КТ, выполняет мультиклассовую сегментацию позвонков по формируемому срединному сагиттальному срезу, определяет аксиальный срез, соответствующий уровню тела выбранного пользователем позвонка. Модуль позволяет уменьшить рутинную нагрузку на рентгенологов во время отбора снимков для дальнейшего анализа. Применение предложенного трёхканального подхода к формированию входных данных для моделей мультиклассовой сегментации позвонков позволило повысить точность сегментации для моделей четырёх архитектур из пяти рассмотренных. Предложенный метод расширения обучающей выборки за счёт пропуска аксиальных срезов оказал положительное влияние на точность моделей с архитектурами ResU-Net и Dense U-Net. Для сегментации позвонков на срединном сагиттальном срезе в разработанном модуле выбрана модель с архитектурой Dense U-Net, обученная на наборе трёхканальных снимков с применением предложенного подхода к расширению выборки. Маски сегментации позвонков, выдаваемые моделью, обрабатываются, после чего используются для поиска нужных аксиальных срезов, а также для формирования обработанных изображений, содержащих срединные сагиттальные срезы. Такие изображения передаются на интерфейс программы для визуализации и анализа медицинских файлов DICOM, с которой интегрирован разработанный модуль, и могут использоваться для контроля результатов его работы. Также на интерфейс программы визуализации выводятся искомые аксиальные срезы, которые могут обрабатываться другими модулями, интегрированными с ней. В рамках дальнейшего развития исследования планируется разрабатывать программные модули для количественного и качественного анализа различных тканей и органов по снимкам КТ, интегрируемые с программой-просмотрщиком.

Список литературы / References

- [1]. Van den Broeck J., Sealy M.J., Brussaard C., Kooijman J., Jager-Wittenaar H., Scafoglieri A. The correlation of muscle quantity and quality between all vertebra levels and level L3, measured with CT: An exploratory study. *Frontiers in Nutrition*, 2023, vol. 10, pp. 1148809. DOI: 10.3389/fnut.2023.1148809.
- [2]. Arayne A.A., Gartrell R., Qiao J., Baird P.N., Yeung J.M. Comparison of CT derived body composition at the thoracic T4 and T12 with lumbar L3 vertebral levels and their utility in patients with rectal cancer. *BMC Cancer*, 2023, vol. 23, no. 1, pp. 56. DOI: 10.1186/s12885-023-10522-0.
- [3]. Recio-Boiles A., Galeas J., Goldwasser B., Sánchez K., Man L., Gentzler R., Gildersleeve J., Hollen P., Gralla R. Enhancing evaluation of sarcopenia in patients with non-small cell lung cancer (NSCLC) by assessing skeletal muscle index (SMI) at the first lumbar (L1) level on routine chest computed tomography (CT). *Supportive Care in Cancer*, 2018, vol. 26, no. 7, pp. 2353-2359. DOI: 10.1007/s00520-018-4051-2.
- [4]. Hong J.H., Hong H., Choi Y.R., Kim D.H., Kim J.Y., Yoon J.H., Yoon S.H. CT analysis of thoracolumbar body composition for estimating whole-body composition. *Insights Imaging*, 2023, vol. 14, no. 1, pp. 69. DOI: 10.1186/s13244-023-01402-z.
- [5]. Теплякова А.Р., Шершнев Р.В., Старков С.О. Метод сегментации мышечной ткани на снимках компьютерной томографии на базе предобработанных трехканальных изображений. *Научно-технический вестник информационных технологий, механики и оптики*, том 24, вып. 4, 2024 г., стр. 661-664. DOI: 10.17586/2226-1494-2024-24-4-661-664. / Teplyakova A.R., Shershnev R.V., Starkov S.O. Method of muscle tissue segmentation in computed tomography images based on preprocessed three-channel images. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2024, vol. 24, no. 4, pp. 661-664 (In Russian). DOI: 10.17586/2226-1494-2024-24-4-661-664.
- [6]. Qadri S.F., Lin H., Shen L., Ahmad M., Qadri S., Khan S., Khan M., Zareen S.S., Akbar M.A., Heyat M.B.B., Qamar S. CT-Based Automatic Spine Segmentation Using Patch-Based Deep Learning. *International Journal of Intelligent Systems*, 2023, vol. 2023, pp. 2345835. DOI: 10.1155/2023/2345835.
- [7]. Khandelwal P., Collins L., Siddiqi K. Spine and Individual Vertebrae Segmentation in Computed Tomography Images Using Geometric Flows and Shape Priors. *Frontiers in Computer Science*, 2021, vol. 3. DOI: 10.3389/fcomp.2021.592296.

- [8]. Cheng P., Yang Y., Yu H., He Y. Automatic vertebrae localization and segmentation in CT with a two-stage Dense-U-Net. *Scientific Reports*, 2021, vol. 11, no. 1. pp. 22156. DOI: 10.1038/s41598-021-01296-18.
- [9]. Saeed M.U., Dikaos N., Dastgir A., Ali G., Hamid M., Hajje F. An Automated Deep Learning Approach for Spine Segmentation and Vertebrae Recognition Using Computed Tomography Images. *Diagnostics*, 2023, vol. 13, no. 1, pp. 2658. DOI: 10.3390/diagnostics13162658.
- [10]. Vania M., Mureja D., Lee D. Automatic spine segmentation from CT images using Convolutional Neural Network via redundant generation of class labels. *Journal of Computational Design and Engineering*, 2019, vol. 6, no. 2, pp. 224-232. DOI: 10.1016/j.jcde.2018.05.002.
- [11]. Qadri S.F., Ai D., Hu G., Ahmad M., Huang Y., Wang Y., Yang J. Automatic Deep Feature Learning via Patch-Based Deep Belief Network for Vertebrae Segmentation in CT Images. *Applied Sciences*, 2019, vol. 9, no. 1, pp. 69. DOI: 10.3390/app9010069.
- [12]. Lu H., Li M., Yu K., Zhang Y., Yu L. Lumbar spine segmentation method based on deep learning. *Journal of Applied Clinical Medical Physics*, 2023, vol. 24, no. 1, pp. e13996. DOI: 10.1002/acm2.13996.
- [13]. Li B., Liu C., Wu S., Li G. Verte-Box: A Novel Convolutional Neural Network for Fully Automatic Segmentation of Vertebrae in CT Image. *Tomography*, 2022, vol. 8, no. 1, pp. 45-58. DOI: 10.3390/tomography8010005.
- [14]. Свидетельство о государственной регистрации программы для ЭВМ. Программа для визуализации и анализа медицинских файлов DICOM, Шершнеv Р. В. (RU), Теплякова А. Р. (RU). № RU2024617422; заявл. 23.03.2024; опубли. 02.04.2024, Бюл. №4. / Shershnev R.V., Teplyakova A.R. Program for visualization and analysis of DICOM medical files. Certificate of state registration of a computer program RU2024617422, registered 23.03.2024, published 02.04.2024 (in Russian).
- [15]. Kaur R., Juneja M., Mandal A. K. A comprehensive review of denoising techniques for abdominal CT images. *Multimedia Tools and Applications*, 2018, vol. 77, no. 17, pp. 22735-22770. DOI: 10.1007/s11042-017-5500-5.
- [16]. Tomczak K., Czerwińska P., Wiznerowicz M. The Cancer Genome Atlas (TCGA): an immeasurable source of knowledge. *Contemporary Oncology*, 2015, vol. 19, no. 1A, pp. A68-A77. DOI: 10.5114/wo.2014.47136.
- [17]. Ronneberger O., Fischer P., Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015, vol. 9351, pp. 234-241. DOI: 10.1007/978-3-319-24574-4_28.
- [18]. Oktay O., Schlemper J., Folgoc L.L., Lee M.J., Heinrich M.P., Misawa K., Mori K., McDonagh S.G., Hammerla N.Y., Kainz B., Glocker B., Rueckert D. Attention U-Net: Learning Where to Look for the Pancreas. *arXiv preprint*, 2018. DOI: 10.48550/arXiv.1804.03999.
- [19]. Diakogiannis F.I., Waldner F., Caccetta P., Wu C. ResUNet-a: a deep learning framework for semantic segmentation of remotely sensed data. *ISPRS Journal of Photogrammetry and Remote Sensing*, 2020, vol. 162, pp. 94-114. DOI: 10.1016/j.isprsjprs.2020.01.013.
- [20]. Pun N. S., Agarwal S. Inception U-Net Architecture for Semantic Segmentation to Identify Nuclei in Microscopy Cell Images. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 2020, vol. 16, no. 1, pp. 12. DOI: 10.1145/3376922.
- [21]. Cai S., Tian Y., Lui H., Zeng H., Wu Y., Chen G. Dense-UNet: a novel multiphoton in vivo cellular image segmentation model based on a convolutional neural network. *Quantitative Imaging in Medicine and Surgery*, 2020, vol. 10, no. 6, pp. 1275-1285. DOI: 10.21037/qims-19-1090.
- [22]. Löffler M.T., Sekuboyina A., Jacob A., Grau A.-L., Scharf A., Hussein M.E., Kallweit M., Zimmer C., Baum T., Kirschke J.S. A Vertebral Segmentation Dataset with Fracture Grading. *Radiology: Artificial Intelligence*, 2020, vol. 2, no. 4. pp. e190138. DOI: 10.1148/ryai.2020190138.
- [23]. Liebl H., Schinz D., Sekuboyina A., Malagutti L., Löffler M.T., Bayat A., El Hussein M., Tetteh G., Grau K., Niederreiter E., Baum T., Wiestler B., Menze B., Braren R., Zimmer C., Kirschke J.S. A computed tomography vertebral segmentation dataset with anatomical variations and multi-vendor scanner data. *Scientific Data*, 2021, vol. 8, no. 1, pp. 284. DOI: 10.1038/s41597-021-01060-0.
- [24]. VerSe: A Vertebrae labelling and segmentation benchmark for multi-detector CT images / A. Sekuboyina, M.E. Hussein, A. Bayat et al. // *Med Image Anal*, 2021, vol. 73, pp. 102166. DOI: 10.1016/j.media.2021.102166.
- [25]. Dataset with segmentations of 117 important anatomical structures in 1228 CT images (Version 2.0.1) [Data set] / J. Wasserthal // *Zenodo*. 2023. DOI: 10.5281/zenodo.10047292.
- [26]. Свидетельство о государственной регистрации программы для ЭВМ. Программа для обучения моделей сегментации мышечной ткани по снимкам компьютерной томографии, Теплякова А. Р.

(RU), Шершнев Р. В. (RU). № RU2024612322; заявл. 18.01.2024; опубл. 31.01.2024, Бюл. №2. / Teplyakova A.R., Shershnev R.V. Program for training muscle tissue segmentation models from computed tomography images. Certificate of state registration of a computer program RU2024612322, registered 18.01.2024, published 31.01.2024 (in Russian).

Информация об авторах / Information about authors

Анастасия Романовна ТЕПЛЯКОВА – старший преподаватель отделения интеллектуальных кибернетических систем Обнинского института атомной энергетики НИЯУ МИФИ. Сфера научных интересов: компьютерное зрение, глубокое обучение, медицинская визуализация.

Anastasia Romanovna TEPLYAKOVA – Senior Lecturer, Obninsk Institute for Nuclear Power Engineering. Research interests: computer vision, deep learning, medical visualization.

Роман Владимирович ШЕРШНЕВ – старший преподаватель отделения интеллектуальных кибернетических систем Обнинского института атомной энергетики НИЯУ МИФИ. Сфера научных интересов: архитектура программного обеспечения, системы поддержки принятия решений, групповой анализ решений.

Roman Vladimirovich SHERSHNEV – Senior Lecturer, Obninsk Institute for Nuclear Power Engineering. Research interests: software architecture, decision support systems, group decision analysis.

